ABSTRACT

Title of dissertation: PLANNING UNDER UNCERTAINTY: MOVING FORWARD

Ugur Kuter, Doctor of Philosophy, 2006

Dissertation directed by: Professor Dana Nau
Department of Computer Science

Reasoning about uncertainty is an essential component of many real-world planning problems, such as robotic and space applications, military operations planning, air and ground traffic control, and manufacturing systems. Planning under uncertainty focuses on how to generate plans that will be executed in environments where actions have nondeterministic effects (i.e., actions may have more than one possible outcome) and the states of the world are not always fully observable. The two predominant approaches for planning under uncertainty are based on Markov Decision Processes (MDPs) and Symbolic Model Checking. Despite the recent advances in these approaches, the problem of how to plan under uncertainty is still very hard: the planning algorithms must reason about more than one possible execution path in the world, and the sizes of the solution plans may grow exponentially. In planning environments that do not admit full observability, the complexity of planning increases by an additional exponential factor since the planner does not know the exact states of the world, and therefore, it must reason over the set of all states that it believes to be in.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Planning Under Uncertainty: Moving Forward	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Department of Computer Science ,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Reasoning about uncertainty is an essential component of many real-world planning problems, such as robotic and space applications, military operations planning, air and ground traffic control, and manufacturing systems. Planning under uncertainty focuses on how to generate plans that will be executed in environments where actions have nondeterministic effects (i.e., actions may have more than one possible outcome) and the states of the world are not always fully observable. The two predominant approaches for planning under uncertainty are based on Markov Decision Processes (MDPs) and Symbolic Model Checking. Despite the recent advances in these approaches, the problem of how to plan under uncertainty is still very hard: the planning algorithms must reason about more than one possible execution path in the world, and the sizes of the solution plans may grow exponentially. In planning environments that do not admit full observability the complexity of planning increases by an additional exponential factor since the planner does not know the exact states of the world, and therefore, it must reason over the set of all states that it believes to be in. This dissertation describes a suite of new planning algorithms for planning under uncertainty with the assumption of full observability. The new algorithms are much more efficient than the previous techniques; in some cases, they find solutions exponentially faster than the previous ones. In particular, our contributions are as follows ? A method to take any forward-chaining classical planning algorithm, and systematically generalize it to work for planning in nondeterministic planning domains, where the likelihood of the possible outcomes of the actions are not known. In our experiments ND-SHOP2, a generalization of the Hierarchical Task Network (HTN) planner SHOP2 [NAI+03], could find solutions in nondeterministic planning domains about two to three orders of magnitude faster than MBP [BCP+01], which uses symbolic model-checking techniques based on Binary Decision Diagrams (BDDs) [Bry92], and which was one of the best previous planners for such domains. ? A way, called ?Forward State-Space Splitting (FS3),? to take the search control (i.e. pruning) technique of any forward-chaining classical planner, such as TLPlan [BK00] TALplanner [KD01], and SHOP2 [NAI+03], and combine it with BDDs. The result of this combination is a suite of new planning algorithms for nondeterministic planning domains. In our experiments, FS3 SHOP2, one of the new algorithms that combines HTNs as in ND-SHOP2 with BDDs as in MBP, was never dominated by either MBP or ND-SHOP2: FS3 SHOP2 could easily deal with problem sizes that neither MBP nor

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**198**	

This dissertation describes a suite of new planning algorithms for planning under uncertainty with the assumption of full observability. The new algorithms are much more efficient than the previous techniques; in some cases, they find solutions exponentially faster than the previous ones. In particular, our contributions are as follows:

- A method to take any forward-chaining classical planning algorithm, and systematically generalize it to work for planning in nondeterministic planning domains, where the likelihood of the possible outcomes of the actions are not known. In our experiments, ND-SHOP2, a generalization of the Hierarchical Task Network (HTN) planner SHOP2 [NAI+03], could find solutions in nondeterministic planning domains about two to three orders of magnitude faster than MBP [BCP+01], which uses symbolic model-checking techniques based on Binary Decision Diagrams (BDDs) [Bry92], and which was one of the best previous planners for such domains.
- A way, called "Forward State-Space Splitting (FS^3)," to take the search control (i.e., pruning) technique of any forward-chaining classical planner, such as TLPlan [BK00], TALplanner [KD01], and SHOP2 [NAI+03], and combine it with BDDs. The result of this combination is a suite of new planning algorithms for nondeterministic planning domains. In our experiments, FS^3_{SHOP2}, one of the new algorithms that combines HTNs as in ND-SHOP2 with BDDs as in MBP, was never dominated by either MBP or ND-SHOP2: FS^3_{SHOP2} could easily deal with problem sizes that neither MBP nor ND-SHOP2 could scale up to, and furthermore, it could solve problems about two or three orders of magnitude faster than the other two.
- A way to incorporate the pruning technique of a forward-chaining classical planner into

the previous algorithms developed for planning with MDPs. The modified algorithms in our experiments were about 10,000 times faster than the original ones on the largest problems the original ones could solve. On another set of problems that were more than 14,000 times larger than the original algorithms could solve, the modified ones took only about 1/3 second.

The new planning techniques described here have good potential to be applicable to other research areas as well. In particular, this dissertation describes such potentials in Reinforcement Learning, Hybrid Systems Control, and Planning with Temporal Uncertainty. Finally, the closing remarks include a discussion on the challenges of using search control in planning under uncertainty and some possible ways to address those challenges.

PLANNING UNDER UNCERTAINTY: MOVING FORWARD

by

Ugur Kuter

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Commmittee:

Professor Dana Nau, Chair/Advisor
Professor Steve Marcus, Dean's Representative
Professor James Hendler
Professor Michael Fu
Professor Atif Memon

To my love, Ferla — for pushing me forward in all those uncertain times

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude and uncountable number of thanks to my advisor, Professor Dana Nau, for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past years. He took a stubborn granite block and carved it out to what Ugur Kuter is today, with his never-lasting patience, understanding, generosity, kindness, and compassion. He is a true master and it was a privilege to be able to be his apprentice during my doctoral training. The experience I had under his supervision will guide me throughout my career and my life.

I express special thanks to Professors Paolo Traverso and Marco Pistore for inviting me to Trento, Italy for an internship and for giving me the opportunity to work there on an important part of this dissertation. They have been unbelievably understanding to my mistakes and helpful in developing my ideas, which constitute an important portion of this dissertation.

I also thank to Professors Jürgen Dix, Michael Fu, Steve Marcus, James Hendler, and John Lemmer, with whom I had the most-rewarding opportunity to work and learn from them regarding research and life.

Finally, I thank to my wife, Ferla, for always being there during all the difficult times of my life in the past 5-6 years. Her strength and her ability to tell me how to pick myself up whenever I fell down always made my life much easier than it would have been, and it will always be so. I also thank to my parents, Güher and Figen Kuter, for

their never-ending support to me in pursuing my goals in life.

In the past years, this research has been supported in part by the following grants: NSF grant IIS0412812, Air Force Research Laboratory F30602-00-2-0505, Naval Research Laboratory N00173021G005, DARPA's REAL initiative, Army Research Laboratory DAAL0197K0135, University of Maryland Institute of Systems Research seed funding, and the FIRB-MIUR project RBNE0195k5, "Knowledge Level Automated Software Engineering." The opinions expressed in this paper are those of the author and do not necessarily reflect the opinions of the funders.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Planning in Artificial Intelligence, Traditionally

Traditional Artificial Intelligence (AI) planning, also known as *classical planning*, requires several restrictive assumptions to be made in the formulations of planning problems. In this view, the environment must contain finitely many objects, and configurations of those objects describe the states of the environment. A classical planner always knows what is true and what is false in a state of the world. Planner's actions have deterministic outcomes; i.e., when executed, an action has a single and an instantaneous effect on the state of the world. Furthermore, the planner's actions are the only cause of change in the world. Thus, the world evolves in discrete and deterministic time steps when a plan (i.e., a sequence of deterministic actions) is executed.

Even under the above restricting assumptions, planning is a hard problem [ENS95]. Over the years, great strides have been made in order to develop efficient techniques for planning. Particularly successful are the techniques that have the ability to use *search-control information*, i.e., auxiliary information that a planner uses during planning in order to guide the planning process. In past international AI planning competitions, planners that can use search-control information consistently worked in most planning domains, solved the most planning problems, and solved them fastest [Bac01, FL02].

Despite these recent advances in classical planning, planning algorithms developed for classical planning problems have not found much applicability in real-world planning problems. The next section overviews some of the reasons why.

1.2 Uncertainty Happens!

Uncertainty happens in the world. When executed, actions may fail to produce their intended outcomes, and/or exogenous events may happen and change the state of the world. Furthermore, the world is not always fully observable to the system executing a plan. Reasoning about such sources of uncertainty is an essential component of many real-world planning problems, such as robotic and space applications, military operations planning, air and ground traffic control, and manufacturing systems. Unfortunately, such applications have characteristics that violate all or most of the restrictive assumptions of classical planning mentioned in the previous section, and as a result, classical planning has been usually limited to simple and toy planning problems.

Classical planning has been extended for reasoning about several forms of uncertainty in real-world planning problems. One of the most widely studied such extensions has been the assumption of *nondeterminism*: in nondeterministic planning environments, actions may have more than one possible outcome when they are executed in the world, and a planner does not know which of those outcomes will actually occur. The nondeterministic outcomes of the actions are used to model possible action failures and/or the effects of exogenous events. Sometimes the planner knows the likelihood of possible action outcomes, in which case, probability distributions over the

possible outcomes of actions are used as a model of uncertainty. In these cases, the primary approach is based on Markov Decision Processes (MDPs); see [BDH99] for an excellent survey of this approach. In MDP planning problems, the objective is to find a *policy* (i.e., a plan expressed as a function that tells which action to perform in each state) that optimizes a utility function. The two basic algorithms for solving MDPs are Value Iteration and Policy Iteration [Ber05]. Other MDP planning techniques have been developed over the years, including factorized MDPs [PPS+02, BDG00], abstraction techniques [DB97, LK02, GK03, DKKN95, HS94], approximation techniques [DKKN93, CKL94, Die00, GKP01b, GKP01a, SP01], symbolic approaches to first order MDPs [BRP01], the use of decision trees and diagrams [HSAHB99, BDG00], generalized linear functions [KP99, KP00, GKP01b], adaptations of heuristic search [HZ98, BL99], and adaptations of greedy search [BG00, BG01b].

There are many real-world planning problems, however, in which probability distributions over the possible outcomes of actions are not available. Examples include applications such as the control of trains and railway stations [CGM+97, CGM+98, CPS+99, CCP+99], and the control of spacecrafts [ACGT01, ACG+01]. In such complex applications, it is very hard to assess the probability distributions either because of the lack of sufficient data to compute those distributions, or because the probabilities are irrelevant since the objective is to guarantee some outcome rather than to make that outcome probable. In either case, the existing planners generate plans that guarantee to achieve some property, when they are executed.

The predominant approach in planning with non-probabilistic models of nondeterminism has been planning as model checking [CPRT03, JVB01, JVB03, Rin02, BCRT01,

PBT01, DPT02]. This approach models the state space basically as a nondeterministic finite-state machine; thus it is like an MDP except that no probabilities are attached to the state transitions, and the objective is to make sure that some property holds in all or some execution paths induced by the state transitions, rather than to optimize a utility function. Solutions to nondeterministic planning problems are classified as weak (at least one execution path will satisfy the goal property), strong (all execution paths will satisfy the goal), and strong-cyclic (all "fair" execution paths will satisfy goals) [CRT98a, CRT98b, DTV99]. [CPRT03] gives a full formal account and an extensive experimental evaluation of planning for these three kinds of solutions.

All of the planning techniques mentioned above are based on the hypothesis that the system that executes the plans (or policies) generated by these techniques can observe the complete state of the world during execution. Under this assumption, the planning systems use complete information about the world states while deciding which action must be planned for which state. There have been some attempts to relax this assumption. An extreme case is planning with *null observability*, where no state information is available about the world [WAS98, SW98]. A more realistic assumption is, perhaps, the *partial-observability* hypothesis: in this case, the system that is responsible for executing that plan interacts with the world through *observations* that provide partial information about the world state, and it attempts to execute the action specified by the plan based on those observations. The same observation may occur in more than one state of the world; thus, the planning algorithms must reason with sets of states in order to generate plans that work over observations. This exponentially increases the search spaces of planning algorithms, which are already huge even under the assumption of full observability.

MDP planning has been extended to the partial-observability case via Partially-Observable MDPs (or POMDPs, for short) [Son78, BG00, CKL94, KLC98, PB00, PB01, Kar01, BDH99]. POMDP planning can be seen as searching over a space of belief states. In its general form, a belief state is defined by a probability distribution over its member states, and planning is done via transformations of these probability distributions. However, this formulation makes the POMDP planning very hard: the number of belief states is huge and in most cases, it may not be finite. As a result, POMDP planning algorithms can only solve very simple and toy planning problems, and they cannot scale up to complex ones. Among the few planning algorithms that have demonstrated some practicality are GPT [BG01a] and PTLplan [Kar01].

Planning as model checking has also been extended to deal with partial observability [BCRT01, BCRT06]. In these works, belief states are defined as a classes of states that represent common observations, and they are compactly implemented by using Binary Decision Diagrams (BDDs) [Bry92]. Planning is done by performing a heuristic search over an AND-OR graph that represents the belief-state space. It has been demonstrated in [BCRT06] that this approach outperformed two other planning algorithms developed for partially-observable nondeterministic domains; namely GPT [BG01a] and BBSP [Rin05].

1.3 Motivation and Contributions

Even under the full-observability hypothesis, planning under uncertainty is a hard problem. In order to generate solution plans (i.e., policies), existing planning algorithms

need to reason about all or most of possible execution paths. This requires exploring all or most of the state space, and the state space can be huge even in some toy planning problems. However, in many planning problems, most of the state space is irrelevant to the solutions for those planning problems, and therefore, can be avoided in the planner's search effort. The inability of the planners to avoid those irrelevant portions of the state space yields their exponential-time behavior in most of the planning domains.

In classical planning domains, on the other hand, many ways have been developed to improve the efficiency of planners by preventing them from visiting unpromising states. This work has been especially successful in forward-chaining planners, such as HSP [BG99], FF [HN01], TLPlan [BK00], TALplanner [KD01], and SHOP2 [NAI+03]. These planners know the current state at all times during planning, which facilitates the use of some powerful pruning techniques. In particular, planners such as TLPlan [BK00], TALplanner [KD01], and SHOP2 [NAI+03] can use very effective pruning techniques since these planners consist of a domain-independent search engine that can make use of domain-specific (but problem-independent) search-control knowledge.

The above observations are the basis of the research described here. In particular, this dissertation describes how to take the forward-chaining based planning techniques originally developed for classical planning, and systematically generalize those techniques for planning under uncertainty in fully-observable planning domains. The generalizations have produced very efficient new planning algorithms compared to the previous techniques developed for planning under uncertainty. The following sections summarize the contributions of this research.

1.3.1 Forward Planning in Nondeterministic Planning Domains

Chapter 3 describes a method to take any forward-chaining planning algorithm developed for classical (i.e., deterministic) planning domains, and systematically generalize it to work in planning domains where actions may have nondeterministic outcomes but no probabilities and utilities associated with them. Section 3.2 first describes an abstract procedure, called FCP, for forward-chaining planning in deterministic domains, and shows that most of the existing forward planners are instances of this abstract procedure. Then, Section 3.3 generalizes FCP to a new abstract planning procedure, called ND-FCP, that has the following property: If a planner P can be described as an instance of FCP, then there is a corresponding instance of ND-FCP that is a "nondeterminization" of P for finding solutions to planning problems in nondeterministic domains. As examples, Section 3.4 provides nondeterminizations of HSP, TLPlan, SHOP2, TALplanner, and the Solve-EBW algorithm described in [GN92].

Section 3.5 presents theorems showing that the generalization technique preserves the correctness properties of the original forward planners. These theorems also show that, under certain conditions, the complexity of the generalized algorithms finding solutions to nondeterministic planning problems are polynomially bounded by those of their original classical versions. In a special case, if the original planning algorithm generates solutions in polynomial time for classical planning problems, then the nondeterminization of that algorithm generates solutions for nondeterministic versions of those problems also in polynomial time.

Section 3.6 presents an experimental evaluation that involves a comparison between

a nondeterminized version of SHOP2, called ND-SHOP2, and the well known MBP planner [BCP+01], in two different planning domains. The experimental results show MBP's CPU time growing exponentially in the size of the problems, confirming the results in [PBT01], and ND-SHOP2's CPU time growing only polynomially. A complexity analysis confirms the experimental results for ND-SHOP2: its running time grows at $O(n^5)$, where n is the size of the problem.

1.3.2 Forward State-Space Splitting in Nondeterministic Domains

Many of the generalized planning algorithms mentioned above are very effective in pruning the search space during planning by using the domain-specific search-control information provided to them. However, if there is no such search-control information available or the available search-control information does not provide effective pruning, the performance of the generalized algorithms such as ND-SHOP2 substantially degrades since those algorithms explore one state at a time during their forward search. On the other hand, planners like MBP are built on symbolic model checking techniques, which enables them to work with abstract collections of states by transforming one such collection into another. This approach has been demonstrated to be very effective in some nondeterministic planning domains, where it efficiently generates solutions without using any search-control information. Therefore, it makes sense to combine the advantages of this approach with those of the planner-generalization method described in the previous section.

Chapter 4 describes *Forward State-Space Splitting (FS^3)*, a way to combine the

pruning technique of any forward-chaining classical planning algorithm, such as TLPlan, SHOP2, and TALplanner, with symbolic model-checking techniques that are based on Binary Decision Diagrams (BDDs). The result of this combination is a suite of new planning algorithms for nondeterministic planning domains. Section 4.5 presents theorems on the correctness, completeness, and termination properties of FS^3. Section 4.7 describes an experimental evaluation of an instance of FS^3, called FS^3_{SHOP2}, that combines Hierarchical Task Network (HTN) decomposition techniques as in ND-SHOP2 with BDD-based representations of planning domains as in MBP. FS^3_{SHOP2} was never dominated by either of MBP or ND-SHOP2, could easily deal with problem sizes that neither the MBP or ND-SHOP2 could scale up to, and furthermore, could solve problems about two or three orders of magnitude faster than the other two.

1.3.3 Forward Planning with Markov Decision Processes (MDPs)

Planning algorithms for MDPs typically have large efficiency problems due to the need to explore all or most of the state space, as discussed previously. Chapter 5 focuses on a way to improve the efficiency of planning on MDPs by adapting the pruning techniques used in forward-chaining classical planners, such as TLPlan and TALplanner in which the search-control knowledge consists of pruning rules written in temporal logic, and HTN planners such as SIPE-2 [Wil90], O-Plan [CT91], and SHOP2 [NAI+03], in which the search-control knowledge consists of HTN task-decomposition templates.

Chapter 5 describes how to modify any forward-chaining MDP planning algorithm, by incorporating into it the search-control algorithm from any forward-chaining plan-

ner. Section 5.2 presents two examples of the new enhanced MDP planning algorithms that have been produced by the modification technique; that is, an enhanced version of the well-known Value Iteration algorithm [Ber05] using the search control rules as in TLPlan and an enhanced RTDP algorithm [BG00, BG03] that uses task decomposition techniques as in SHOP2.

Section 5.3 describes conditions under which the enhanced MDP planning algorithms are guaranteed to find optimal answers, and conditions under which they can do so exponentially faster than the original ones. The experimental results presented in Section 5.4 demonstrate that the enhanced algorithms running exponentially faster than the original ones. On the largest problems the original algorithms could solve, the modified ones ran about 10,000 times faster. In only about 1/3 second, the modified algorithms could solve problems whose state spaces were more than 14,000 times larger.

1.4 A Note on Planning Forward, Forward, and Forward

Previous techniques to planning under uncertainty usually focused on general ways of solving planning problems in uncertain environments. Although these techniques have been very useful for understanding the characteristics of planning problems under uncertainty, their practicality has been limited to simple planning problems since they are not able to solve the planning problems efficiently and they cannot scale to large problems.

This dissertation, in its entirety, deals with a new approach for planning under uncertainty; that is, it develops new planning algorithms for uncertain environments by *taking a class of classical planning techniques and generalizing them to work for planning*

under uncertainty. The particular focus here is on those classical planning techniques that do forward-chaining search, where the search starts from the initial states of a planning problem and continues toward the goal states. The planners that perform forward-chaining search know the current state of the world at all times during that search, which enables them to exploit effective and expressive search control techniques based on the information from the state of the world.

The rationale behind using search control in classical planning is the observation that not all possible execution paths in a planning domain are relevant to the solutions of a planning problem in that domain, and therefore, those irrelevant execution paths can be eliminated during planning. The most popular way of eliminating irrelevant execution paths is to specify what to do and/or what not to do in a planning domain in order to generate solutions. The classical planning algorithms that have the ability to use such search-control information have been the most successful, and they have been demonstrated to solve complex and large classical planning problems.

Many planning problems in uncertain environments share the above property with classical planning problems. That is, although planning algorithms must examine more than one execution path in order to generate solutions in uncertain environments, most of the possible execution paths are irrelevant to those solutions. As a result, most of the techniques developed for search control in classical planning, when generalized to work in uncertain environments, provide the same sort of efficiency improvements for planning under uncertainty as they have done in classical planning. This dissertation describes several ways to do such generalizations.

The approach taken in this dissertation is not limited to planning under uncertainty.

In that regard, this dissertation concludes by describing ways on how to do similar generalizations of classical search-control techniques to work for synthesizing controllers for hybrid systems, reinforcement learning, and planning under temporal uncertainty.

Chapter 2

Preliminaries

2.1 Classical Planning

Classical planning is usually formalized by starting with the definition of a first-order language L and augmenting this language with additional symbols and expressions [GNT04]. In the language L, we have a finite number of predicate and constant symbols, and no function symbols, i.e., every term is either a variable symbol or a constant symbol. We use the standard definitions for logical atoms and literals.

A *state* is a set (i.e., a conjunction) of ground atoms in L. Intuitively, a state describes the atoms that are true in the world. Here, we use the well-known *closed-world assumption* that any logical atom that is not specified by a state is assumed to be false in that state of the world. A state s satisfies a positive ground atom l (i.e., a ground atom), if $l \in s$. Otherwise, s does not satisfy l. Similarly, a state s satisfies a negative literal $\neg l$ if $l \notin s$. Otherwise, s satisfies $\neg l$.

A *planning operator* is an expression of the form $o = (h\ pre\ del\ add)$. h is an expression of the form $op(x_1, \ldots, x_k)$ such that op is an *operator symbol* and x_i are logical terms. pre, the *preconditions* of the planning operator o, is a set (i.e., a conjunction) of literals. del and add, the *delete-list* and the *add-list* of o, are sets of logical atoms. Intuitively, the delete-list of o describes the set of atoms to be deleted from the state of

the world if the operator o is applied in that state. Similarly, the add-list of o describes the atoms to be added in the current state of the world. An *action* is an ground instance of a planning operator.

A *classical planning domain description* (or a *classical planning domain*, for short) is a deterministic state-transition system $\Sigma = (S, A, \gamma)$ where S and A are the finite sets of states and the actions, respectively, and γ, the state-transition function, is a function defined as

$$\gamma : S \times A \rightarrow 2^S,$$

such that given a state $s \in S$ and action $a \in A$, $|\gamma(s, a)| \leq 1$. Note that this formalizes the *determinism* assumption of classical planning that each action has only one possible outcome. If $\gamma(s, a) = \emptyset$ then we say that the action a is not applicable in the state s. A *plan* in a classical planning domain is a sequence of actions $\langle a_0, a_1, a_2, \ldots, a_k \rangle$ such that, when executed in a state s_0, $\gamma(s_i, a_i) = \{s_{i+1}\}$ for each $i = 0 \ldots k$.

A *classical planning problem description* (or a *classical planning problem*, for short) in a domain $\Sigma = (S, G, \gamma)$ is a tuple $P = (\Sigma, s_0, G)$ where $s_0 \in S$ is the initial state and $G \subseteq S$ is the set of goal states. A *solution* is a plan π such that, when executed in the initial state s_0, π reaches to a goal state $s_{k+1} \in G$.

State-space search is one of the basic methods for generating solution plans for classical planning problems. In this method, the search space is a subset of the state space – i.e., the search space is a subset of all possible states in a classical planning domain –, and it is usually generated either by a forward-chaining or by a backward-chaining search, both can be characterized as a form of the well-known backtracking search [HS78].

```
Procedure FCS(s, G, π)
  if s ∈ G then return(π)
  applicable ← {(s, a) | a is an action and γ(s, a) ≠ ∅}
  if applicable = ∅ then return(FAILURE)
  nondeterministically choose (s, a) ∈ applicable
  return( FCS(γ(s, a), G, π ∪ {(s, a)}) )
```

Figure 2.1: An abstract forward-chaining procedure for state-space search in classical planning. In the initial call of the procedure, π is the empty plan and s is the initial state.

In forward state-space search, the search process starts with the initial state of a classical planning problem and continues by successively generating successor states by applying actions in the current state until the goal state is generated. Figure 2.1 shows an abstract procedure FCS that describes this search process. In the current state s, FCS first generates the set of all actions applicable in s and then chooses one of those actions, say a, nondeterministically and continues the search process from the successor state $\gamma(s, a)$.

In FCS, backtracking occurs when there is no action applicable to the current state s – i.e. $applicable = \emptyset$ in the Figure 2.1. In this case, the algorithm backtracks to the previous state and explores another search branch by choosing another action applicable in that state – if any.

Backward-chaining state space planning is similar to the forward version described above, except that the search process starts at a goal state and uses the inverse state-transition function γ^{-1} in order to generate the predecessor state of a state given an action – i.e. $\gamma^{-1}(s', a) = s$, if $\gamma(s, a) = s'$. The search continues until the initial state is generated in this manner. A backward search procedure has a similar backtracking point compared to its forward counterpart: the procedure backtracks when there is no action that can generate the current state when applied in some other state.

2.2 Search Control in Planning

Some of the most impressive recent advances in classical planning are based on the use of heuristics for organizing the search space and controlling the planning process. Sometimes the heuristics are *domain-independent*, i.e., intended for use in many different planning domains, and sometimes they are *domain-specific*, i.e., tailored to a specific problem domain. Domain-independent heuristics specify general problem-solving principles that work in every planning domain the heuristic functions are used. Domain-specific heuristics, on the other hand, specify search-control knowledge specialized for a particular planning domain and does not usually provide any benefits in another.

The trade-off between using domain-independent and domain-specific heuristics is the straightforward one: flexibility vs. efficiency. Although domain-independent heuristics are much more flexible for they can be used in many planning domains, the latter provides much more effective search-control in the particular domains they are designed for. As a result, the planning algorithms that are able to use domain-specific heuristics solve the planning problems much more efficiently than the ones that exploit domain-independent heuristics, and they can scale up to larger planning problems than the others.

The subsequent sections present an overview of the existing domain-independent and domain-specific heuristic techniques developed for classical planning.

2.2.1 Domain-Independent Search Control

Probably the most successful and therefore the most popular way of generating domain-independent heuristics is based on *problem relaxation* techniques. Problem re-

laxation involves ruling out some of the constraints from the definition of the planning problems in a domain in order to create "relaxed" versions of those problems. The planning algorithms, then, solve these relaxed problems and use their solutions as heuristic information for controlling their search in the original ones. This approach have been very successful in many algorithms for classical planning, including the works reported in [BF97, Koe99, KS98a, KS98b, BG99, HG00, HBG05].

A popular technique to derive heuristics from relaxed versions of planning problems is the use of *planning graphs*, i.e., data structures that compactly represent the set of solutions to a relaxed planning problem. The planning algorithms that use this technique generate a planning graph for the input planning problem by performing a forward search from the initial states toward the goal states, while ignoring the constraints on the possible interactions of actions in the plans. Then, the algorithms use the planning graph generated in this way to constrain their search for solutions for the original planning problem. Planning graphs were first introduced as tools to guide the planning process in the GraphPlan algorithm [BF97], and their success led to the development of many successors, including IPP [Koe99], BlackBox [KS98a, KS98b], and AltAlt [NKN02].

Reachability analysis based on generating distance (or cost) estimates in relaxed versions of planning problems is another way of deriving domain-independent heuristics in classical planning [BG99, HN01, NK01, YS02]. The planning algorithms that use this approach usually perform a forward or backward sweep of the search space induced by a relaxed version of a planning problem generated by ignoring the negative effects of actions (i.e., by ignoring the action effects that delete some information from the state of the world) and/or by assuming some independence properties over action preconditions.

Such processing of the search spaces of planning problems are similar to finding shortest-paths over graphs [CLRS01], and they enable the planners to compute the estimates for distances or costs of states and/or atoms to the goals of the relaxed planning problem. Those estimates are then used to guide the order in which the planners visit nodes in the search space of the original planning problem.

Planning graphs and distance-based reachability analysis can be combined to provide better heuristics for solving classical planning problems. The FF planner [HN01] and its successors are good examples for this hybrid approach that has been successful in a large number of planning domains as demonstrated in past several international planning competitions [Bac01, FL02].

2.2.2 Domain-Specific Search Control

When *domain-specific* heuristics are used, sometimes the planners themselves are domain-specific, i.e., they work only in a particular domain such as process planning [HSMN96] or the game "Bridge" [SNT98]. In other cases, the planner consists of a domain-independent planning engine (which is usually a forward-chaining state-space search engine), plus a language for writing domain-specific problem-solving knowledge.

In some cases, the language for writing domain-specific search-control knowledge is based on modal temporal logic formulas (e.g., TLPlan [BK00] and TALplanner [KD01]). These formulas specify some properties of the solution plans for the input planning problem that should and/or should not hold during the execution of those plans. In other words, they specify *acceptable* behaviors of sequences of action (i.e., plans) gener-

Figure 2.2: An unstack action in the Blocks World domain, with blocks.

ated in a planning domain. For example, a search-control formula may require a condition be ALWAYS hold in every state that is generated during the execution of a plan. Another may require a condition be EVENTUALLY hold in some state that is to be generated during execution. Finally, a formula may require a condition be hold in the NEXT state that arises by applying an action in the current state. Such search-control formulas are used by TLPlan and TALplanner to control their search during planning by avoiding action sequences that do not satisfy a given search-control formula.

For example, consider the well known Blocks World planning domain, where there are a number of blocks in the world; some are on top of another block and the others are on the table. There are four actions: pickup the block from the table, putdown the block on the table, stack the block on top of another block, if there are no blocks on the latter, and unstack the block from the top of another block, if there are no blocks on the former. Figure 2.2 shows an instance of this planning domain and the operation of an unstack action in that instance.

[BK00] reports a search-control formula for TLPlan in the Blocks World domain

as follows:

$$\begin{aligned}\chi: \quad \Box(\quad & \forall[?x : clear(?x)]goodtower(?x) \Rightarrow \odot(clear(?x) \\ & \qquad \vee \exists[?y : on(?y, ?x)]goodtower(?y)) \\ & \wedge badtower(?x) \Rightarrow \odot(\neg\exists[?y : on(?y, ?x)]) \\ & \wedge (on(?x, table) \\ & \quad \wedge \exists[?y : GOAL(on(?x, ?y))] \\ & \quad \wedge \neg goodtower(?y)) \Rightarrow \odot(\neg holding(?x))).\end{aligned}$$

In the above formula, $?x$ and $?y$ are variable symbols that represent individual blocks in the domain. This formula characterizes the following two facts. First, the planner must place a block only on *good towers*, but not on *bad towers*. A *good tower* is a stack of blocks in which every block is in its goal position. A *bad tower* is a tower that is not a good tower. The first two conjuncts of the above formula deals with this case. Second, there may also be some useless actions applicable in the current state that can be pruned out. For example, suppose we have a block $?x$ that is intended to be on block $?y$ but is currently on the table, and $?y$ is on top of a bad tower in the current state. In this case, there is no point in moving $?x$ on top of $?y$ right now since it will be moved back again when the planner tries to build a good tower under $?y$. The last conjunct of the formula models this fact.

Another widely-recognized formalization for specifying domain-specific heuristics for planning is based on *Hierarchical Task Networks (HTNs)* (e.g., SHOP2 [NAI+03], SIPE-2 [Wil88], and O-PLAN [CT91]). Here, the domain-specific knowledge is encoded by means of *tasks* (i.e., symbolic representations of real-world activities), and *task-*

decomposition methods that specify the possible ways of decomposing those tasks into smaller ones. Planning starts by the input tasks to be achieved, along with specifications on possible orders they should be achieved, and proceeds by decomposing those tasks into smaller tasks until only primitive tasks that can directly be executed in the world is left. The primitive tasks along with their ordering constraints constitute a solution plan for the input planning problem. If task decomposition fails at some point during planning, then this means that there is no solution plan for the input planning problem given the input tasks and task-decomposition procedures. In that case, the planning algorithms return failure.

An example for tasks and task-decomposition methods as in SHOP2 [NAI+03] is shown in Figure 2.3 for the Blocks World domain. This figure shows a task move-block for moving blocks from one location to another in the world and a task-decomposition method for accomplishing this task. The argument of the task move-block specifies those blocks that have been already moved to their goal locations. Intuitively, this task-decomposition method encodes the following search-control heuristic for SHOP2:

- if there is an "unsolved" block b such that b can be moved to its goal position, then do the following:
  - if b is on top of a block and its goal position is on some other block that is on top of a good tower, then move b to its goal position, mark it as "solved", and continue with other "unsolved" blocks;
  - else if b is on top of a block and its goal position is on the table, then move it, mark it as "solved", and continue with other "unsolved" blocks;

```
(:method (move-block ?solved)
 ;; method for moving x from y to z
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
        (goal (on ?x ?z)) (different ?x ?z) (clear ?z) (not (need-to-move ?z)))
 ((!unstack ?x ?y) (!stack ?x ?z) (move-block (?x . ?solved)))

 ;; method for moving x from y to table
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
         (goal (on-table ?x)))
 ((!unstack ?x ?y) (!putdown ?x table) (move-block (?x . ?solved)))

 ;; method for moving x from table to y
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on-table ?x)
        (goal (on ?x ?y)) (clear ?y) (not (need-to-move ?y)))
 ((!pickup ?x) (!stack ?x ?y) (move-block (?x . ?solved)))

 ;; method for moving x out of the way
 ((arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
  (need-to-move ?x))
 ((!unstack ?x ?y) (!putdown ?x table) (move-block ?solved))

 ;; if nothing else matches, then we're done
 nil
 nil)
```

Figure 2.3: A task-decomposition method that describes the search-control information for SHOP2 in the Blocks World domain.

  - else if b is on the table and its goal position is on a block that is on top of a good tower, then move it, mark it as "solved", and continue with other "unsolved" blocks;

- else if there is an "unsolved" block that cannot be moved to its goal position, then move it on the table and continue with other "unsolved" blocks;
- else, there is no "unsolved" block left.

Although it can take some effort to write and tune domain-specific search-control

heuristics, the case for doing so is quite strong. All of the planners mentioned above have greater expressive power than the existing domain-independent planners; e.g., they can call attached procedures, can do axiomatic inference and numeric computations, and can reason about durative actions whose durations are not fixed in advance. In past AI planning competitions [Bac01, FL02], these planners were the fastest, could solve the hardest problems, and could handle the widest variety of planning domains.

2.3 Planning under Uncertainty

In many real-world planning problems, a planner does not know the exact outcomes of its actions when it executes them in the world; e.g., an action may fail to achieve its intended outcomes, and/or there may be some exogenous events that may change the state of the world without the control of the system that executes the plan. Furthermore, the system that executes a plan may not be able to observe the state of the world that arises from executing an action. *Planning under uncertainty* focuses on the problem of how to plan in environments that admit the above sources of uncertainty. In some cases, the objective of planning under uncertainty is to generate plans that satisfy a given property throughout all or some execution paths in the world; in others, it is to generate plans that optimize some expected utility when they are executed. It is hard to achieve these objectives: the algorithms must reason about all or most of the possible execution paths, and the sizes of the solution plans may grow exponentially.

This section reviews the two most widely known and used approaches to planning under uncertainty under the assumption of fully observability. The presentation here par-

ticularly focuses on the existing techniques that have been very successful to reduce the complexity of planning under uncertainty in practice. Other related works on planning under uncertainty that are relevant to this dissertation are reviewed in Chapter 6.

2.3.1 Planning based on Markov Decision Processes (MDPs)

Planning based on MDPs is the primary approach for solving planning problems that deals with nondeterminism, probabilities, costs, and rewards (see [BDH99] for a survey).

An *MDP description of a planning problem* (or an *MDP planning problem*, for short) is a tuple (Σ, S_0, G) where Σ is an *MDP description of a planning domain*, $S_0 \subseteq S$ is the set of initial states, and $G \subseteq S$ is the subset of the goal states.

An *MDP description of a planning domain* (or an *MDP planning domain*, for short) is a tuple of the form $\Sigma = (S, A, \gamma, \alpha, Pr, C)$ where S is a finite set of all possible states and A is a finite set of all possible actions. $\gamma : S \times A \rightarrow 2^S$ is the state transition function. Note that we do not have the determinism requirement in this formulation; i.e., $|\gamma(s, a)|$ does not have to be ≤ 1. The set of applicable actions in a state s is $\mathsf{app}(s) = \{a \mid a \in A \text{ and } \gamma(s, a) \neq \emptyset\}$.

In Σ, $0 < \alpha \leq 1$ is the discount factor. Pr is the transition-probability function defined as

$$Pr : S \times A \times S \rightarrow [0, 1]$$

such that

$$\sum_{s' \in \gamma(s,a)} Pr(s, a, s') = 1, \ \forall s \in S, a \in \mathsf{app}(s),$$

and C is the cost function defined as

$$C : S \times A \rightarrow \Re^{+}.$$

In MDP planning, the objective is to find a *policy* (i.e., a plan expressed as a function that tells which action to perform in each state) that optimizes a utility function. More formally, a policy π is defined as a partial function

$$\pi : S \rightarrow A.$$

The reason that a policy π need not be a total function is that if a state $s \in S$ is not reachable from the initial states in S_0 by successively applying the actions in A, then s can safely be omitted from the domain of π.

In an alternative set-theoretic view, a policy can be seen as a set of state-action pairs of the form (s, a) such that $\gamma(s, a) \neq \emptyset$ — i.e., such that the action a is applicable in the state s. In this view, the set of states S_π in a policy is the set $\{s \mid (s, a) \in \pi\}$. This dissertation uses the above set-theoretic representations of policies.

Given a policy π, the *value function* $V^\pi(s)$ is the expected sum of the future discounted costs, i.e.,

$$V^\pi(s) = E_\pi[\sum_{t \geq 0} \alpha^t \, C(s_t, \pi(s_t)) \, | s_0 = s],$$

where s_t is the state of the MDP at time t, and $E_\pi[\cdot]$ is understood with respect to the search space induced by the state transition function γ and the transition probabilities. A *solution* for an MDP planning problem is a policy π^* such that when executed in an initial state $s_0 \in S_0$, π^* reaches a goal state in G with probability 1, and no other policy π' has both the same property and a lower expected cost. An MDP planning problem is *solvable* if and only if there is a solution for it.

It is well-known [Put94] that the optimal value $V(s)$ for state s can be computed by solving the system of equations

$$V(s) = \begin{cases} 0, & \text{if } s \in G \\ \min_{a \in \mathsf{app}(s)} Q(s,a), & \text{otherwise,} \end{cases} \tag{2.1}$$

$$Q(s,a) = C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s')V(s'). \tag{2.2}$$

Solution Methods for MDP Planning

The two basic algorithms for solving MDP planning problems are Value Iteration and Policy Iteration. The former is a *dynamic programming* technique that computes optimal values of states in a backwards fashion by computing the value of the current state using the cost function C and the values of the all possible successor states. This computation is based on Bellman's *Principle of Optimality*; which says, "an optimal policy must have the property that whatever the initial state and the initial decision (for planning an action in that state) are the remaining states and the decisions (for planning actions in those states) must constitute an optimal policy with regard to the states resulting from the first decision." [Ber05]. Figure 2.4 shows the pseudocode for the Value Iteration algorithm for generating optimal solutions for MDP planning problems.

The Policy Iteration algorithm performs a search over the space of all possible policies. The algorithm successively alternates the following two phases: the policy-evaluation phase and the policy-update phase. In the former phase, Policy Iteration computes the expected values of the states in the current policy by solving the system of equations shown in Equations 2.1 and 2.2. In the policy-update phase, the algorithm re-

```
Procedure Value Iteration
 select any initialization for the value function V
 while V has not converged do
  π ← ∅
  for every state s ∈ S
   for every a ∈ app(s)
    Q(s,a) ← C(s,a) + α Σ_{s'∈γ(s,a)} Pr(s,a,s') V(s')
    V(s) ← min_{a∈app(s)} Q(s,a)
    a ← argmin_{a∈app(s)} Q(s,a)
    π ← π ∪ {(s,a)}
 return π
```

Figure 2.4: The Value Iteration algorithm. In the procedure above, S is the state space in an MDP planning problem.

fines a policy to a new policy where the initial states of the new one have smaller expected costs. The Policy Iteration algorithm terminates when there are no further refinements are possible, in which case the current policy is an optimal policy (i.e., a solution) for the input planning problem.

One limitation of the Value Iteration and Policy Iteration algorithms is their high computational complexity due to the need for examining the entire state space (or large portions of it) to generate optimal policies. For complex planning problems the state space can be quite huge: for planning problems expressed using probabilistic STRIPS operators [HM93, KHW94] or 2TBNs [HSAHB99, BG96], planning is EXPTIME-hard [Lit97].

Real-Time Dynamic Programming (RTDP) has emerged as a promising technique for MDP planning [BG00, BG03]. Figure 2.5 shows the pseudocode of the RTDP planning procedure. RTDP is a planning algorithm based on *real-time* forward search, which is performed by simulating possible execution paths in the world, rather than by exploring all or most of the states that can be reached from the initial states of the input MDP

Procedure RTDP
 select any admissible initialization for V
 while V has not converged relative to a parameter ϵ do
  $s \leftarrow s_0$
  while $s \notin G$ do
   $a \leftarrow \text{argmin}_{a \in \text{app}(s)} Q(s,a)$
   $V(s) \leftarrow C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s')V(s')$
   pick $s' \in \gamma(s,a)$ with probability $Pr(s,a,s')$
   $s \leftarrow s'$
 extract the greedy optimal policy π given V and s_0
 return π

Figure 2.5: The Real-Time Dynamic Programming, RTDP, procedure.

planning problem. In each iteration of the outer while loop, RTDP performs a greedy search going forward starting from the initial state towards the goal states of the input MDP planning problem.[1] RTDP's forward search at each iteration is a stochastic simulation of the greedy partial policy. The planning procedure updates the values of the states visited in an iteration using the Bellman update

$$V(s) \leftarrow C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s')V(s'),$$

where a is the greedy action (i.e., the current best action that has the minimum $Q(s,a)$ value) in a state s that the algorithm is currently exploring. After updating the value of s, the planning algorithm simulates the execution of a in s by stochastically selecting a single successor state s' of s based on the state-transition function $Pr(s,a,s')$. The

[1] Without loss of generality, RTDP assumes that there is a single initial state in the description of MDP planning problems [BG00, BG03]. Note that MDP planning problems with multiple initial states can easily be modeled as a planning problem with a single initial state and a special action such that applying that action in the initial state generates a set of successor states that correspond to the initial states of the original planning problem.

simulation continues with the state s' until a goal state is generated.

Experimental comparisons of RTDP with traditional dynamic programming approaches such as Value Iteration showed that RTDP is able scale up to much larger MDP planning problems than Value Iteration. However, RTDP is not an optimal algorithm as Value Iteration is — i.e., it does not guarantee to generate optimal solutions for every MDP planning problem; in some cases, it even does not guarantee to terminate [BG03]. If the initial estimate of the value function V is not over-estimating (i.e., $V \leq E_{\pi^*}[\cdot]$) and there is a path with positive probability from every state in an MDP planning domain Σ to the goal states of the input planning problem, then the algorithm is shown to terminate after finitely many forward searches and return an optimal solution [BG03].

2.3.2 Planning as Model Checking

Planning as Model Checking deals with the problem of planning under nondeterminism and partial-observability. The primary difference between MDP planning problems and the planning problems here is that planning as model checking does not require probabilities, rewards, and costs to be known, and the objective is to satisfy a property in all or some execution paths induced by a policy rather than to optimize some utility function. This planning approach is useful in applications where transition probabilities are unavailable due to lack of data, or where the transition probabilities are irrelevant because the objective is to guarantee some outcome rather than to make that outcome probable. Applications that are being investigated include the control of trains and railway stations [CGM+97, CGM+98, CPS+99, CCP+99], and the control of spacecraft

[ACGT01, ACG+01].

Planning as model checking uses *nondeterministic models* for formalizing planning domains. A *nondeterministic description of a planning domain* (or a *nondeterministic planning domain*, for short) is given in terms of a nondeterministic state-transition system $\Sigma = (S, A, \gamma)$ where S and A are the finite sets of all possible states and actions in the domain, and γ, the state-transition function, is a function defined as

$$\gamma : S \times A \rightarrow 2^S.$$

An action a is applicable in a state s if $\gamma(s, a) \neq \emptyset$. The set S_a of all states in which a is applicable is $\{s \mid s \in S \text{ and } \gamma(s, a) \neq \emptyset\}$.

As in MDP planning, a *policy* π is defined as a partial function from states to actions. The set S_π of states in a policy π is $\{s \mid (s, a) \in \pi\}$. The set S_π^t of *terminal states* of π is $\{s' \mid (s, a) \in \pi, s' \in \gamma(s, a), \text{ and } s' \notin S_\pi\}$.

The *execution structure* Σ_π induced by a policy π is the subsystem of the particular planning domain Σ, defined as $\Sigma_\pi = (V_\pi, E_\pi)$ such that V_π is the set of the nodes of Σ_π. Each node in V_π represents either a state in the policy π or a terminal state of π — i.e., $V_\pi = S_\pi \cup S_\pi^t$. E_π is the set of arcs between the nodes of Σ_π, which represent possible state transitions caused by actions in π.

The notion of reachability in execution structures can be formalized as follows. Let π be a policy, and let $\Sigma_\pi = (V_\pi, E_\pi)$ be the execution structure induced by π. For any two nodes $s, s' \in V_\pi$, s is a π-*ancestor* of s' in Σ_π if there is a path in Σ_π from s to s'. Similarly, s' is called a π-*descendant* of s in Σ_π.

A *dead-end* state s of π is a state in the execution structure Σ_π such that

- if s a terminal state of π and there is no action applicable in s, or
- if s is a non-terminal state of π and s has no π-descendants in the terminal states of Σ_π that are not dead-end states.

A state-action pair (s, a) in π is a *dead-end* state-action pair if all of the states in $\gamma(s, a)$ are dead-end states.

A *planning problem description in a nondeterministic planning domain* (or a *nondeterministic planning problem*, for short) is a tuple of the form (Σ, S_0, G) where $\Sigma = (S, A, \gamma)$ is a nondeterministic planning domain, $S_0 \subseteq S$ is the set of initial states, and $G \subseteq S$ is the set of goal states. Solutions to planning problems are classified as *weak* (at least one execution trace will reach a goal), *strong* (all execution traces will reach goals), and *strong-cyclic* (all "fair" execution traces will reach goals) [CRT98a, CRT98b, DTV99]. More precisely,

- A *weak solution* to a nondeterministic planning problem is a policy π such that if for every initial state s in S_0, there exists at least one path in Σ_π that starts from the node that represents s and reaches to a final node that represents a goal state. A policy π is a *candidate* weak solution if, for each initial state $s \in S_0$, there exists at least one path in Σ_π that starts from s and ends in a terminal state in S^t_π.
- A *strong solution* is a policy π such that (1) if every finite path in Σ_π reaches to a final node that satisfies the goals, and (2) there are no infinite paths (i.e., no cycles) in Σ_π. A policy π is a *candidate* strong solution if (1) every finite path in Σ_π reaches to a terminal state in S^t_π, and (2) there are no cyclic paths in Σ_π.
- A *strong-cyclic solution* is a policy that is guaranteed to achieve the goals of the plan-

ning problem under a so-called "fairness assumption" [CPRT03]; which says, the execution of the policy must guarantee to reach to the goal states, if every cyclic path in Σ_π is executed only finitely many times in the world. This means that a policy π is a strong-cyclic solution if every (finite or infinite) path in Σ_π can be extended to a finite execution path that reaches to a goal state. A policy π is a *candidate* strong-cyclic solution if every path in Σ_π can be extended to a finite execution path that reaches to a state in S^t_π.

A nondeterministic planning problem is *solvable* if it has any of the three kinds of solutions described above.

Solution Methods for Planning as Model Checking

The model-checking based planning algorithms for generating weak, strong, and strong-cyclic solutions use breadth-first search techniques that proceed backwards starting from the goal states towards the initial states of a planning problem. The primary and the most important difference between these planning algorithms and traditional backward-chaining state-space planning, as described in Section 2.1, is that the former perform the backward search over sets of states, whereas the latter explores one individual state at a time. As a result, the model-checking based planning algorithms are able to explore huge state spaces that traditional backward search could not scale up to.

The backward search over clusters of states is performed using Preimage functions as search primitives. Given a set of states S, a Preimage function computes some set of predecessors of S in a single backward operation. Model-checking based planning

techniques use primarily two types of Preimage functions: namely, the WeakPreimage and StrongPreimage functions. The WeakPreimage of a set S of states is defined as follows [CPRT03]:

$$\mathsf{WeakPreimage}(S) = \{(s,a) \mid \gamma(s,a) \neq \emptyset \text{ and } \gamma(s,a) \cap S \neq \emptyset\}.$$

Intuitively, the WeakPreimage of a set S of states include every state s in Σ such that at least one of the successor states generated by applying an action a in s is in S.

The StrongPreimage of S is defined as follows [CPRT03]:

$$\mathsf{StrongPreimage}(S) = \{(s,a) \mid \gamma(s,a) \neq \emptyset \text{ and } \gamma(s,a) \subseteq S\}.$$

Intuitively, the StrongPreimage of a set S of states includes every state s in a planning domain Σ such that all of the successor states that are generated by applying a in s are in S.

The model-checking based planning algorithms are based on either of these Preimage functions, depending on whether their objective is to generate weak, strong, and strong-cyclic solutions. [CPRT03] gives an extensive overview of these algorithms. Weak planning simply consists of successive WeakPreimage computations, starting from the goal states towards the initial states. The algorithm terminates when for each initial state, it generates a path that reaches to a goal state. The weak planning algorithm is a special case of strong and strong-cyclic planning algorithms, and therefore, it is not described here in detail.

Strong planning is done by performing a backward search that starts from the goal states of a planning problem and the empty policy, and by extending the current policy with the actions generated as a result of performing successive StrongPreimage

```
Procedure StrongPlan(S_0, G)
  π ← FAILURE; π' ← ∅
  while π' ≠ π and S_0 ⊈ (G ∪ S_π') do
    preimage ← StrongPreimage(G ∪ S_π)
    π'' ← PruneStates(preimage, G ∪ S_π)
    π ← π'; π' ← π' ∪ π''
  if S_0 ⊆ (G ∪ S_π) then return(MkDet(π))
  return(FAILURE)
```

Figure 2.6: The StrongPlan algorithm for generating strong solutions in nondeterministic planning domains.

operations. Figure 2.6 shows, StrongPlan, the strong planning algorithm described in [GNT04].[2] The strong planning process ends either when the initial states of the input planning problem are reached, or when there are no possible further extensions to the current policy. The former case marks the successful termination of planning, and therefore, the algorithm returns the generated policy as a solution for the input planning problem. In the latter case, however, the planning process fails to generate a solution for that planning problem.

In StrongPlan, the function PruneStates removes any state-action pair (s, a) if the current partial policy π already specifies another action for s. The function MkDet "determinizes" the final policy π: it returns a policy $\pi' \subseteq \pi$ such that $S_\pi = S_{\pi'}$ and for every state $s \in S_{\pi'}$ there exists only one state-action pair (s, a) in π'. The formal definitions of all of these functions are given in [CPRT03, GNT04].

Similar to strong planning, strong-cyclic planning is also based on backward breadth-first search over sets of states using Preimage operations. Figure 2.7 shows

[2] [CPRT03] gives a more detailed pseudocode for strong planning; however, the exposition in [GNT04] is simpler and easier to understand.

```
Procedure StrongCyclicPlan(S_0, G)
  π ← ∅; π' ← UnivPol
  while π ≠ π' do
    π ← π'
    π'' ← PruneOutgoing(π', G)
    π''' ← ∅
    repeat
      X ← π'''
      π''' ← π'' ∩ WeakPreimage(G ∪ S_π''')
    until X = π'''
  π' ← π'''
  if S_0 ⊆ (G ∪ S_π') then
    return(MkDet(RemoveNonProgress(π, G)))
  return(FAILURE)
```

Figure 2.7: The StrongCyclicPlan algorithm for generating strong-cyclic solutions in nondeterministic planning domains.

StrongCyclicPlan, the strong-cyclic planning algorithm described in [GNT04]. Strong-cyclic planning differs from the strong planning algorithm in the way it exploits backward search as well as in its preimage operations. The planning procedure starts from the *universal policy*, i.e., the policy that contains all of the possible state-action pairs in the given planning domain, and successively eliminates state-action pairs from this policy. Each iteration of the strong-cyclic planning algorithm performs a backward search starting from the goal states toward the initial states of the input planning problem. During this backward search, StrongCyclicPlan identifies and eliminates the state-action pairs that does not specify any progress toward the goal states of the planning problem. In order to identify such state-action pairs, a more relaxed preimage operation than StrongPreimage is needed; in particular, StrongCyclicPlan uses the WeakPreimage function described above. WeakPreimage operations ensure that a state-action pair (s, a) will not be eliminated in this iteration if there is at least one possible execution path that starts from s

and leads toward to a goal state by applying a in s. All of the state-action pairs that do not specify any progress toward the goal states are eliminated from the current policy in this iteration. Note that elimination of state-action pairs in one iteration may require other state-action pairs be eliminated in the successive iterations.

Strong-cyclic planning continues until no further elimination is possible. Note that, at this point, the planning procedure reaches to a fixpoint policy π by successively eliminating state-action pairs from the universal policy it started with. This fixpoint policy induces an execution structure in which every path can be extended to finite execution path that reaches to the goal states. Thus, a final correctness check is needed to make sure that π also includes the initial states of the input planning problem as well. If so, StrongCyclicPlan first removes the redundant state-action pairs from π that do not make any progress towards the goals. Then it "determinizes" π as described above and returns the final policy as solution for input planning problem. Otherwise, the planning process terminates with a failure to generate a solution for that planning problem.

In StrongCyclicPlan, the functions PruneStates and MkDet are as explained above. The functions RemoveNonProgress and PruneOutgoing in StrongCyclicPlan are responsible for ensuring that no state-action pair is left in the final solution policy that needs to be removed. The formal definitions of all of these functions are given in [CPRT03, GNT04].

Both StrongPlan and StrongCyclicPlan planning procedures described above have been theoretically shown to be sound and complete planning algorithms. They are sound in the sense that if they return a solution for the given planning problem, that solution is guaranteed to be a strong or strong-cyclic solution for that problem, respectively.

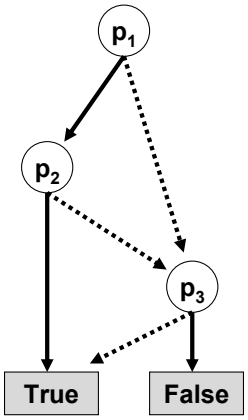

Figure 2.8: A BDD representation of the propositional formula $(p_1 \wedge p_2) \vee \neg p_3$. The solid arrows represent the case where a proposition p_i is TRUE and the dotted arrows represents the case where p_i is FALSE.

They are complete in the sense that if they return failure, then there exists no strong or strong-cyclic solution to the given planning problem, respectively.

2.4 Binary Decision Diagrams (BDDs) in Planning

As we discussed in the previous section, the model-checking based algorithms perform their backward searches over clusters of states, rather than over individual states as in the traditional state-space planning. This allows for using *symbolic model-checking* techniques to compactly represent sets of states in those algorithms and implementing the search procedures over those compact representations. It has been experimentally demonstrated that in some cases, the use of such compact state representations exponentially improves the run times required by those planning algorithms to generate solutions to planning problems [BCP$^+$01, PBT01, JVB03].

The most popular symbolic model-checking method for compactly representing

sets of states during planning is based on the use of *Ordered Binary Decision Diagrams* (or BDDs for short) [Bry92]. BDDs are data structures that provide a canonical form for representing Boolean functions (i.e., propositional logical formulas). More specifically, a BDD is a directed acyclic graph in which the terminal nodes (i.e., nodes that do not have any outgoing edges) represent the logical values TRUE and FALSE, and the non-terminal nodes represent the propositions in a Boolean formula. Each nonterminal node has two children BDDs. The truth value of a propositional formula represented as a BDD is given by the terminal node that is reached by traversing the BDD starting from the root node and ending at that terminal node. As an example, Figure 2.8 shows the BDD representation of the following propositional formula:

$$(p_1 \wedge p_2) \vee \neg p_3,$$

where each p_i is a proposition. In this example, suppose p_1 is FALSE, and p_2 and p_3 are TRUE, so the entire formula is FALSE. The evaluation of the formula to this fact is done over the BDD of Figure 2.8 is as follows. We start from the p_1 node. Since p_1 is FALSE, we follow the dotted arrow out of this node, coming to the p_3 node. Since p_3 is TRUE, we follow the solid arrow out of p_3 and end up in the FALSE node. Note that this is correct since $\neg p_3$ is FALSE, and therefore, the entire formula is FALSE.

As demonstrated above, BDDs can be combined to compute the negation, conjunction, and disjunction of propositional formulas. The combination of two BDDs, say b_1 and b_2, can be performed in linear time $O(|b_1|\ |b_2|)$ where $|b_i|$ is the size of the BDD b_i — i.e., $|b_i|$ is the number of variables (nonterminal nodes) in the BDD b_i [Bry92].

The model-checking based planning algorithms discussed in the previous section

exploit BDDs to implement sets of states over which the backward searches are performed and the transformations over BDDs through negation, conjunction and disjunction to implement the Preimage operations over those sets of states. This way, the transformation of a set of states into its preimage is done in a single BDD transformation operation, which, in some cases, provides a very efficient way to solve planning problems.

Unfortunately, there is no guarantee that in the general case, BDD-based compact representations of states provide huge performance gains in the planning algorithms that use such representations. The reason is that the ordering of the propositions represented by the nonterminal nodes of a BDD plays a crucial role in determining the truth value of the propositional formula represented by the entire BDD since the performance of traversing the graph structure of a BDD depends on the compactness of that structure itself. As demonstrated in several experimental studies [PBT01, CPRT03, KN04], there are many planning problems in which the structure of the BDDs is lost due to successive transformations performed over them during planning. In such cases, the planning algorithms do not benefit from using compact state representations at all; they perform very poorly even on the simplest toy planning problems.

Dynamic variable ordering techniques have been developed and used in BDD-based planning algorithms to address these drawbacks [Rud93, PSP94]. However, restructuring the BDDs is itself a costly computation and does not provide any benefits if it is done every time a transformation is performed over the BDDs. Identifying the exact conditions for restructuring the BDDs during planning is also a hard problem, and often, those conditions depend on the planning problems that is being solved. Thus, dynamic variable ordering is not always helpful.

Chapter 3

Forward-Chaining Planning in Nondeterministic Planning Domains

The previous chapter mentioned some efficient classical planning algorithms such as SHOP2 [NAI+03], TLPlan [BK00], TALplanner [KD01], and gave some examples of the search-control information that these algorithms can use. This chapter describes a method to take any forward classical planning algorithm (e.g., HSP, SHOP2, TLPlan, and TALplanner), and systematically generalize it to work in nondeterministic planning domains – i.e., planning domains where actions may have more than one possible outcome. Such generalizations enable us to exploit many of the desirable characteristics of these planners, such as the ability to use search-control information, to achieve highly efficient planning in nondeterministic domains.

Section 3.1 first describes a formalization for "nondeterministic versions" of classical planning problems. Sections 3.2 to 3.4 describe the generalization method in detail. Section 3.5 presents theorems showing that generalizations preserve the correctness properties of the original classical planners. These theorems also show that, under certain conditions, the complexity of our generalized algorithms for finding solutions to planning problems in nondeterministic domains are polynomially bounded by those of their original classical versions. As a special case, if the original planning algorithm generates solutions in polynomial times for a classical planning problem, then its corresponding generalization generates solutions for nondeterministic versions of those problems also in

polynomial times.

The theoretical results are confirmed by an experimental comparison of one of the generalized algorithms, ND-SHOP2, a generalization of SHOP2, with the state-of-the-art planner MBP [BCP+01] originally developed for nondeterministic domains. On problems where the branching factor in the search spaces are very high, the well-known MBP algorithm took exponential times, confirming prior results by others. On those problems, ND-SHOP2, on the other hand, took only polynomial times. We confirm the polynomial-time figures by a complexity analysis.

3.1 Deterministic vs. Nondeterministic Planning Problems

A classical description of a planning domain $\Sigma = (S, A, \gamma)$ assumes that $|\gamma(s, a)| \leq 1$ for any state-action pair, in order to formalize the requirement that actions have deterministic effects in classical planning. In a nondeterministic planning domain description, this determinism requirement is relaxed by lifting the constraint on the size of γ for modeling one or more possible outcomes of an action.

The relaxation of γ in nondeterministic planning domains is the basis of a relationship between the classical planning domains and their nondeterministic versions. Let $\Sigma = (S, A, \gamma)$ be a classical planning domain. Then a nondeterministic planning domain $\Sigma' = (S', A', \gamma')$ is a *nondeterministic version* of Σ, if the following holds:

- $S = S'$; and
- there is a one-to-one mapping det from A' to A such that the following holds:
  - for each state $s \in S$, if $\gamma'(s, a') = \emptyset$ then $\gamma(s, det(a')) = \emptyset$.

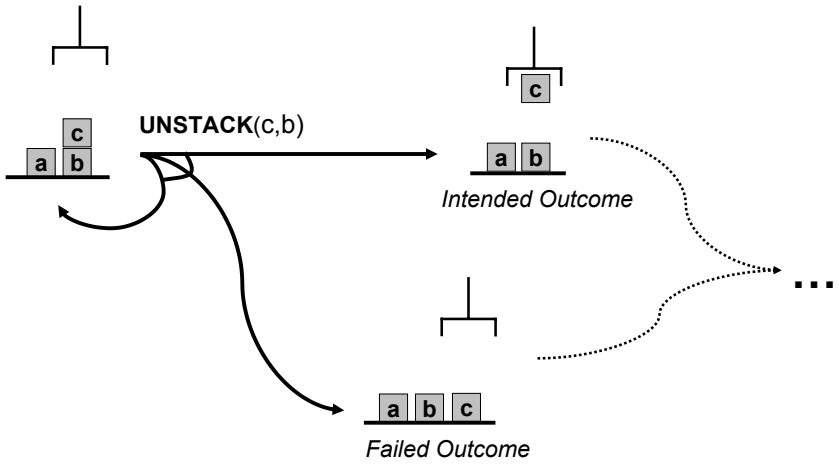

Figure 3.1: A nondeterministic unstack action in a nondeterministic version of Blocks World with 3 blocks.

- Otherwise, $\gamma(s, det(a')) \in \gamma'(s, a')$.

Intuitively, a nondeterministic planning domain is a nondeterministic version of a classical one if both descriptions specify the same set of states and actions, except that the actions in the former may have additional effects in the planning domain.

As an example, consider the classical Blocks World domain, as described in Section 2.2. A nondeterministic version of Blocks World contains the same state space and the same set of actions as the original domain does, except that an action in this version may have its *intended* outcome that is the same outcome it has in the classical case but it may also fail to have any effects in the world or it may drop the block on the table (e.g., in the case the gripper is slippery). Figure 3.1 gives an illustration of the nondeterministic unstack action with a failure in the nondeterministic version of a Blocks World domain with three blocks.

A nondeterministic planning problem $P' = (\Sigma', S_0, G)$ is a nondeterministic ver-

```
Procedure FCP(s, G, π, χ)
  if s ∈ G then return(π)
  actions ← {a ∈ A | γ(s, a) ≠ ∅ and
                     acceptable(s, a, χ) holds}
  if actions = ∅ then return(failure)
  nondeterministically choose a ∈ actions
  s' ← result(s, a)
  χ' ← progress(s, a, χ)
  π' ← append(s, a, π)
  return(FCP(s', G, π', χ'))
```

Figure 3.2: An abstract version of a forward-chaining classical planning algorithm. In the initial call of the procedure, π is the empty plan, s is the initial state, and χ is the initial search-control information.

sion of a classical planning problem $P = (\Sigma, s_0, G)$, if Σ' is a nondeterministic version of Σ and $s_0 \in S_0$. Note that both P' and P have the same set of goals according to this definition.

3.2 FCP: An Abstract Procedure for Forward-Chaining Planning

Section 2.1 discussed the backtracking forward-chaining state-space search: the search procedure FCS starts at an initial state and proceeds by successively generating new states. A successor of a state s is generated by nondeterministically choosing an action a in s and applying it in s. The search terminates if a goal state is generated in this way and the search trace starting from the initial state and ending at the goal state describes a solution to the input planning problem. FCS performs very poorly in most of the classical planning problems since it generates an search space that is exponential in the sizes of those problems. Section 2.2 described the use of search-control heuristics for classical planning that enable planners to explore smaller search spaces than FCS does.

Figure 3.2 shows the abstract FCP planning procedure that formalizes a class of forward state-space planning algorithms that have the ability to use search-control heuristics. In this pseudocode, s is the current state, G is the goal, and π is the current partial plan. In the initial call of the procedure, s is the initial state of the input classical planning problem and π is the empty plan – i.e., $\pi = \emptyset$. Given an initial state s and a set of goal states G, FCP searches for a sequence of actions — i.e., a plan π —, which generates a goal state in G, when π's actions are executed in s in the order they are specified.

In FCP, χ, the *search-control information*, is any auxiliary information available to the planner that specifies one or more actions applicable in a state of the world, among all possible alternatives. FCP uses this information in its *search-control function* acceptable, in order to determine whether an action a should or should not be considered in a state. The formal definitions for both χ and acceptable depend on the particular planning algorithm, and we will discuss several planning algorithms that are instances of FCP and their respective search-control mechanisms in the subsequent section.

If there is no acceptable actions for the state s given the auxiliary information χ, then FCP backtracks and tries other possibilities in the previous iterations of the search process — note that, although there might be an action a that is *applicable* in s, i.e., $\gamma(s, a) \neq \emptyset$, FCP may prune out all of them, given the search-control information χ. If FCP has an acceptable action a for a state s, it generates (1) the state s' that arises from applying a in s and (2) the search-control information χ' that is to be used along with s'. The functions result and progress are responsible for these tasks, respectively. As before, the formal definitions of these functions depend on the particular instance of FCP and examples are given in the subsequent section.

3.2.1 Instances of FCP

This section presents several instances of the abstract FCP planning procedure, which include examples of classical planning algorithms that use domain-independent and domain-specific search-control information.

Heuristic Search Planner (HSP).

HSP [BG99, BG01b] is a combination of *hill-climbing search* (i.e., *greedy local search*) and a variation of the A^* search algorithm [RN03]. The planner is a simple forward state-space search engine that incorporates a family of both admissible and non-admissible domain-independent heuristics for controlling its search.

HSP uses domain-independent search-control information during planning as follows. Given a planning problem, HSP compiles the search-control information during planning by computing "distance/cost estimates" of the goal states from a state generated during the planning process. The planner obtains a distance/cost estimate of a state s to a goal state by solving a "relaxed" version of the original classical planning problem, where the negative effects (i.e., the delete-lists) of the actions are ignored. The optimal cost of solving the relaxed problem from the state s is a lower bound on the cost of solving the original problem from s [BG99].

The acceptable function in HSP is responsible for computing the heuristic costs. In each state s generated during planning, acceptable performs a forward search in order to generate the cost $h(s)$ of solving the relaxed planning problem from the state s. The planner then chooses the action a that, when applied in s, generates the next state s'

with the least-cost $h(s')$ value that is less than the cost value computed for s — i.e., $h(s') \leq h(s)$. Then, the search continue from the state s' until a goal state is generated in this way.

Selecting a locally best next state and searching from that state on towards the goal states is demonstrated to be an effective technique for reaching goal states efficiently; however, it suffers from the *search plateaus* in the topology of the search space of most planning problems, a well-known problem for most hill-climbing search algorithms [RN03]. A *search plateau* is a portion of the search space that consists of states whose heuristic values do not change. In other words, the planner generates and visits a sequence of states such that $h(s') = h(s)$ for any two successor state s and s' in that sequence. A hill-climbing search algorithm caught within a search plateau does not have any information whether or not the planning process is proceeding towards the goals. To prevent spending unnecessary computational time in a search plateau, usually search algorithms make a random move or restart their searches after a prespecified number of moves in a plateau [RN03]; HSP does the latter.

The result function in HSP is responsible for generating the next state s' by applying the best action a in a state s, given the current heuristic function of HSP. Furthermore, this function also implements the restarting mechanism HSP uses to escape from a search plateau — i.e., result simply returns the initial state of the input planning problem, if the algorithm makes more than a prespecified moves without improving the heuristic value of the states visited.

In HSP, the progress function shown in Figure 3.2 does not have any functionality since HSP computes the search-control information (i.e., the heuristic cost functions)

```
Procedure TLPlan(s, G, π, χ)
 if s ∈ G then return(π)
 actions ← {a ∈ A | γ(s, a) ≠ ∅}
 if actions = ∅ then return(failure)
 nondeterministically choose a ∈ actions
 s' ← result(s, a)
 χ' ← Progress(s, χ)
 if χ' = FALSE then return FAILURE
 π' ← append(s, a, π)
 return(TLPlan(s', G, π', χ'))
```

Figure 3.3: The TLPlan planning algorithm. In the initial call, π is the empty plan, s is the initial state, and χ is the initial temporal-logic formula.

itself in the acceptable function.

TLplan and TALplanner.

TLPlan [BK00] and TALplanner [KD01] are two planning algorithms that have the ability to use domain-specific search-control information.[1] TLPlan and TALplanner have been very successful in solving classical planning problems in many experimental studies [Bac01, FL02]. Both are instances of the abstract FCP planning procedure. Figures 3.3 and 3.4 show the pseudocodes of TLPlan and TALplanner, respectively.

In TLPlan and TALplanner, the search-control information χ is specified in terms of a logical formula written in some form of modal temporal logics, as described in Sec-

[1] [KD01] describes two versions of TALplanner, called the *sequential* and the *concurrent* TALplanner. The former is developed for solving planning problems in classical planning domains, whereas the latter extends the sequential algorithm to incorporate reasoning about action durations and plans that contain concurrent actions whose executions may overlap. This dissertation considers only the sequential TALplanner since it does not concern with temporal characteristics of actions.

```
Procedure TALplanner(N_s, N_G, π, N_χ)
 if N_s satisfies N_G then return(π)
 actions ← {a | a is an action in N_A, and a is applicable in s}
 if actions = ∅ then return(FAILURE)
 nondeterministically choose a ∈ actions
 N_s' ← result(N_s, a)
 N_χ' ← Progress(N_s, a, N_χ)
 if SearchControl(N_s, N_χ') = FALSE
  then return(FAILURE)
 π ← append(N_s, a, π)
 return(TALplanner(N_s', N_G, π', N_chi'))
```

Figure 3.4: The TALplanner planning algorithm. In the initial call, π is the empty plan, $\mathcal{N}_s$ is the TAL formula that describes the initial state s, $\mathcal{N}_G$ is the TAL formula that describes the goal states G, and $\mathcal{N}_\chi$ is the TAL formula that describes the initial search-control formula.

tion 2.2. In particular, TLPlan uses *Linear Temporal Logic (LTL)* and TALplanner uses the *Temporal Action Language (TAL)*.[2] Both TLPlan and TALplanner use temporal-logic formulas written in these languages to specify acceptable behaviors of action sequences and they avoid any action sequence that does not satisfy a formula during the search. The acceptable function in TLPlan and TALplanner checks whether an action a, when applied in a state s, violates the current search-control formula χ. An action a that is applicable in a state s violates χ if it generates a successor state s' in which χ does not

[2]Note that, although the description of TALplanner in [KD01] seems that the planning algorithm performs a search in a space of TAL formulas, the internal planing engine is a simple forward-chaining state-space search algorithm that starts with the initial state and searches over a state space described by the input planning domain. Given a TAL formula $\mathcal{N}$ that specifies a planning domain, a planning problem, and a set of search-control rules for this domain, one can always split $\mathcal{N}$ into pieces such as $\mathcal{N}_G$, $\mathcal{N}_A$, $\mathcal{N}_\chi$, and $\mathcal{N}_s$, which correspond to the descriptions of the goal states, the actions, the initial search-control information, and the initial state of the input planning problem, respectively.

hold. Otherwise a is acceptable to be applied in s.

If there are no acceptable actions to be applied in the current state s, then both algorithms backtrack and try other alternative actions in the previous steps of the planning process. Otherwise, they generate the set of acceptable actions in a state s during their search. They nondeterministically choose one of those actions and generate the successor state s' that is the result of applying that action in s. Then, the planners generate the search-control formula χ' to be used in s'. The progress function is responsible for this task, and it is a direct implementation of the *Progress* algorithms defined for TLPlan and TALplanner in [BK00] and [KD01], respectively.

Starting from an initial state and an initial search-control formula, TLPlan and TALplanner successively generate new states and search-control formulas associated with those states until a goal state is reached. At that point, the sequence of actions, i.e., the plan, generated by the planners is a solution for the input planning problem and the planners return this plan.

SHOP2.

Figure 3.5 shows the pseudocode for the SHOP2 algorithm. In SHOP2, the objective of the planner is to accomplish *tasks*; i.e., symbolic representations of activities to be performed in the world. Tasks can be either *primitive* or *nonprimitive*. A primitive task can be directly executed in the world, whereas a nonprimitive task needs to be decomposed into smaller tasks (or subtasks), each of which can be either primitive or nonprimitive. A *task network* is a set of tasks and a set of constraints on the order that

```
Procedure SHOP2(s, G, π, w, M)
 if w is the empty task network then return(π)
 T ← {t | t ∈ w and t has no predecessors}
 nondeterministically choose a task t ∈ T
 if t is a primitive task then
  actions ← {(a, σ) | a is an action, σ is a substitution s.t.
                      head(a) = σ(t), and a is applicable in s}
  if actions = ∅ then return(FAILURE)
  choose (a, σ) ∈ actions
  w' ← σ(w − {t})
  π ← append(s, a, π)
  s ← γ(s, a)
 else
  methods ← {(m, σ) | m is an instance of a method in M, σ is a
                      substitution s.t. head(m) = σ(t), and
                      m is applicable in s}
  if methods = ∅ then return(FAILURE)
  choose (m, σ) ∈ methods
  w' ← ApplyMethod(s, w, t, m, σ)
 return(SHOP2(s, G, π, w', M))
```

Figure 3.5: The SHOP2 planning algorithm. In the initial call, π is the empty plan, s is the initial state, w is the initial task network. and M is the set of available task-decomposition methods.

those tasks must be achieved.

Given a task network w and a state, SHOP2 decomposes the nonprimitive tasks in w into smaller and smaller tasks in the order those tasks will be achieved in the world. For this purpose, SHOP2 uses *task-decomposition methods*, which are "operational procedures" to specify possible ways to decompose a task into its subtasks [NAI+05]. Task decomposition in SHOP2 provides a way for controlling the planner's search by focusing the search to only those sequences of primitive tasks (i.e., actions) that are solution prefixes.

In SHOP2, the search-control information χ is a pair of the form (w, M) where w

is the current task network and M is the set of all task-decomposition methods available to the planner. Given a state s and the current search-control information χ, the acceptable function holds for an action a that is applicable in s such that (i) a appears in some task network w' that is produced by recursively decomposing tasks in the task network w specified in χ, and (ii) a has no predecessors in w'. More specifically, suppose t is the current task selected by the current invocation of the algorithm, as shown in Figure 3.5. If t is a primitive task and there is an action a that accomplishes t then the acceptable function holds for an action a in the state s of this invocation. If t is a nonprimitive task then SHOP2 deomposes t into smaller and smaller subtasks until a primitive task is generated. Then, acceptable holds for the action that accomplishes that primitive task.

Once SHOP2 computes all the acceptable actions in s given χ, it nondeterministically chooses one of them and generates the state s' that arises from applying that action in s. The search-control information to be used with s' is produced via the progress function: progress(s, a, χ) is the pair (w'', M) such that w'' is the task network produced by removing a from w', the task network that satisfies the conditions for a being acceptable in s as described above. More specifically, as shown in Figure 3.5, if the current task t is primitive then w'' is the task network produced by removing t from w. Otherwise, w'' is the task network produced by replacing t by its subtasks specified by the task-decomposition method being used for t.

The planning process terminates when there are no nonprimitive tasks left to further decompose, and in that case, the set of primitive tasks along with their ordering constraints is returned as a solution plan for the input planning problem.

Solve-EBW.

The Solve-EBW algorithm [GN92] is a domain-specific planning algorithm developed for solving planning problems in the Blocks-World domain. Starting from an initial state (i.e., a configuration of blocks), Solve-EBW enters a loop in which it attempts to move a clear block to its goal position. If there are no such blocks that can be moved to their goal positions, the algorithm arbitrarily moves a clear block to the table. The planning process continues until each block in the world is at its goal position. It has been shown that this algorithm can solve Blocks-World problems in lower-order polynomial times in the number of blocks in the domain [GN92, ST01].

It is rather straightforward to show that Solve-EBW is an instance of the FCP procedure. It is a direct implementation of FCP, such that the search-control information χ specifies a block that can be moved in the current state into its goal position. The search-control function acceptable specifies the particular move action for χ – i.e., acceptable specifies whether the block χ should be picked up from the table, put down on the table, stacked on another block, or unstacked from the top of another block. The progress function specifies the next block to be moved in the world.

3.3 ND-FCP: Nondeterminized FCP

This section describes a general method for taking classical forward-chaining planners and *nondeterminizing* them, i.e ., translating them into planners that find weak, strong, and strong-cyclic solutions for nondeterministic planning problems. The basis of this method is the FCP planning procedure for classical planning domains described

above, and the corresponding procedure ND-FCP for planning in nondeterministic planning domains. The following discussion presents the nondeterminization of FCP for strong-cyclic planning since this is the most general form of solutions in nondeterministic planning problems as described in the previous chapter. Once the nondeterminized version of FCP for strong-cyclic planning is established, it is somewhat straightforward to modify this nondeterminized planning procedure to produce weak and strong nondeterminizations.

The abstract FCP planning procedure generates a solution plan (or equivalently, an execution path) from the initial states of the input classical planning problem to the goal states. The ND-FCP procedure for strong-cyclic planning in nondeterministic domains is similar to FCP, except that ND-FCP includes some additional bookkeeping operations. These bookkeeping operations deal with multiple possible outcomes of actions and with executions paths induced by the policies that may or may not violate the "fairness assumption" on the strong-cyclic solutions, as described in Section 2.3.2.

Figure 3.6 shows the ND-FCP procedure for strong-cyclic planning in nondeterministic domains. In this figure, the underlines indicate how the coding from FCP is embedded in ND-FCP. In particular, ND-FCP generalizes the forward OR-search of FCP to a forward AND-OR search, in which an AND branch corresponds to the different possible outcomes of applying an action in a state, and an OR-branch corresponds to the different choices of actions applicable — more precisely, to the different choices from the acceptable actions — in a state. To implement this, ND-FCP uses an $OPEN$ set, which contains a set of states and the search-control information associated with them. The states in $OPEN$ are those states generated by ND-FCP during its forward search,

```
Procedure ND-FCP(OPEN, G, π, solved)
 if OPEN = ∅ then return(π)
 select a pair (s, χ) ∈ OPEN and remove it
 if s ∈ G then solved ← solved ∪ {s}
 else if s ∉ S_π then
  actions ← {a ∈ A | result(s, a) ≠ ∅ and acceptable(s, a, χ) holds}
  if actions = ∅ then return(FAILURE)
  nondeterministically choose a ∈ actions
  χ' ← progress(s, a, χ)
  π' ← append(π, ⟨(s, a)⟩)
  π ← π'
  OPEN' ← OPEN ∪ {(s', χ') | s' ∈ result(s, a)}
 else if s does not have a π-descendant in (StatesOf(OPEN) ∪ solved) \ S_π then
  return(FAILURE)
 return(ND-FCP(OPEN', G, π, solved)
```

Figure 3.6: ND-FCP, the nondeterminization of FCP for finding strong-cyclic solutions for nondeterministic planning problems. In the initial call of the procedure, π is the empty policy, $solved$ is the empty set, $OPEN$ is a set of pairs of the form (s, χ) where s is an initial state and χ is the initial search-control information. The underlines indicate how the coding from FCP is embedded in ND-FCP.

but no actions are specified by the current partial policy for them.

ND-FCP also uses a $solved$ set, which is a set of goal states that are reached by the planning procedure during its forward search. The $solved$ set is used to ensure that a policy returned by ND-FCP is a solution for the input nondeterministic planning problem as described below.

At each recursive invocation, ND-FCP first selects a pair (s, χ) from $OPEN$. If s is a goal state, then ND-FCP inserts s into its $solved$ set and continues from other open states in $OPEN$. Otherwise, if ND-FCP has not already planned an action for it — i.e., $s \notin S_\pi$ —, then the procedure generates the set of actions that are applicable in s by using the current search-control information χ and the search-control function

acceptable. Then, it chooses one of those applicable actions, say a, and generates the set of successor states of s by applying a in s. ND-FCP also generates the successor search-control information to be used in each of those successor states by using the progress function. Note that this is exactly the same as in FCP, except that the result function now returns more than one possible successor state that arises from applying a in s. The successor pairs of states and the corresponding search-control information are inserted into the $OPEN$ set.

If s is already among the states of the current partial policy π, then the procedure performs a further π-descendancy check to decide whether or not the current partial policy is a candidate strong-cyclic solution for the input planning problem. This π-descendancy check is done as follows: if the current state s has a π-descendant in the currently $solved$ states then the cycle induced by the current state s does not violate the "fairness assumption" in strong-cyclic planning. Similarly, if s has a π-descendant in the current $OPEN$ states that have not been visited before, then ND-FCP cannot eliminate the current partial policy at this point, since there is still a possibility that during the planning process, the π-descendant of s in $OPEN$ will be expanded to reach to a goal state. On the other hand, if s does not satisfy these checks, then the cycle induced by s in the current partial policy violates the conditions of strong-cyclic solutions, and therefore, ND-FCP returns FAILURE and backtracks in order to try other search branches in the search space.

If the π-descendancy checks described above does not return FAILURE, ND-FCP continues planning by recursively calling itself with the new $OPEN'$ set that contains the new pairs of states and the corresponding search-control information generated in this invocation. The planning process terminates when no open states left to explore — i.e.,

$OPEN = \emptyset$. In that case, ND-FCP returns the policy π. As shown in the next section, π is indeed a solution for the input nondeterministic planning problem that was given to ND-FCP, and ND-FCP will return such a solution policy if there exists one.

So far, ND-FCP is described as an abstract planning procedure for generating strong-cyclic solutions for nondeterministic planning problems. If a nondeterministic planning problem admits strong or weak solutions, it is straightforward to modify the ND-FCP procedure in Figure 3.6 to generate a strong or a weak solution for that planning problem. For strong planning, the π-descendancy check in ND-FCP needs to removed and the planning procedure is modified to return FAILURE as soon as it detects a cyclic execution trace in the current partial policy. Weak planning requires a further modification to the algorithm; namely, whenever ND-FCP generates goal state, it needs to remove every situation (s, χ) that s is not an initial situation from $OPEN$. This ensures that the planning algorithm generates an execution path from each initial state to a goal state.

3.4 Instances of ND-FCP

The nondeterminization technique described above specifies how to take a class of forward planners — namely, those planning algorithms that are instances of FCP —, and generalize them to work in nondeterministic planning domains. In many cases, it is possible to use/generalize the search-control information produced for an original classical planner to work for its corresponding nondeterminized version. Note that, however, the nondeterminization technique itself does not describe how to transfer the search-control information χ used by the instances of FCP in classical (i.e., deterministic) planning do-

```
Procedure ND-TLPlan(OPEN, G, π, solved)
 if OPEN = ∅ then return(π)
 select a pair (s, χ) ∈ OPEN and remove it
 if s ∈ G then solved ← solved ∪ {s}
 else if s ∉ S_π
  actions ← {a ∈ A | a is applicable in s}
  if actions = ∅ then return(FAILURE)
  nondeterministically choose a ∈ actions
  χ' ← Progress(s, χ)
  if χ = FALSE then return(FAILURE)
  π' ← append(s, a, π)
  π ← π'
  OPEN' ← OPEN ∪ {(s', χ') | s' ∈ γ(s, a)}
 else if s does not have a π-descendant in (StatesOf(OPEN) ∪ solved) \ S_π then
  return(FAILURE)
 return(ND-TLPlan(OPEN', G, π, solved))
```

Figure 3.7: ND-TLPlan, the nondeterminization of TLPlan for finding strong-cyclic solutions for nondeterministic planning problems. The underlines indicate how the coding from TLPlan is embedded in ND-TLPlan.

mains over nondeterministic settings. The reason is that different instances of FCP use different formalizations to describe the search-control information they use; therefore, the way to generalize the search-control information in an instance FCP depends on the way such information is formalized in that particular planner. This section addresses this issue by describing several instances of ND-FCP and possible ways to use the search-control information developed for the original classical planners in the nondeterminized ones.

3.4.1 ND-TLPlan

Figure 3.7 shows the nondeterminized version of TLPlan, called ND-TLPlan. Note that, since TLPlan is a simple forward-chaining search algorithm that is an instance of FCP, ND-TLPlan is mostly a direct implementation of the abstract ND-FCP planning

procedure.

In most cases, the search-control information χ that is written in *Linear Temporal Logic (LTL)* for a classical planning domain can be modified to be used by ND-TLPlan in nondeterministic versions of that classical planning domain. As an example, consider again the classical Blocks World domain and its nondeterministic version as we discussed previously in Section 3.1. Consider the search-control formula described for TLPlan in Section 2.2:

$$\begin{aligned}\chi: \quad \Box(\quad & \forall[?x : clear(?x)]goodtower(?x) \Rightarrow \odot(clear(?x) \\ & \qquad \vee \exists[?y : on(?y, ?x)]goodtower(?y)) \\ & \wedge badtower(?x) \Rightarrow \odot(\neg\exists[?y : on(?y, ?x)]) \\ & \wedge (on(?x, table) \\ & \quad \wedge \exists[?y : GOAL(on(?x, ?y))] \\ & \quad \wedge \neg goodtower(?y)) \Rightarrow \odot(\neg holding(?x))).\end{aligned}$$

This search-control formula can be used in the nondeterministic version of Blocks World by incorporating in it a *failure-recovery strategy* as follows. Each time an action fails in the world by dropping the block on the table, we immediately pick that block up before executing any other action. This strategy yields solution policies whose sizes are polynomial in the size of a solution plan for the classical version of the Blocks World domain, since each time an action fails, the pickup action immediately takes the planner to an intended state.

The above failure-recovery strategy can be encoded by modifying the search-control formula χ as follows. For each action in the nondeterministic Blocks World do-

main, we specify the corresponding failure-recovery condition as follows. For example, for the unstack action shown previously in Figure 3.1, we have

$$\begin{aligned}
\chi_{fail}^{unstack} &: on(?x, ?y) \wedge clear(?x) \wedge handempty \wedge \exists[?z : GOAL(on(?x, ?z))] \\
&\quad \wedge \odot (ontable(?x)) \\
\chi_{recover}^{unstack} &: \chi_{fail}^{unstack} \wedge \odot \odot (holding(?x)).
\end{aligned}$$

The first formula above specifies the condition when the unstack action fails. The second formula specifies the condition that must be satisfied during planning when such a failure occurs. More specifically, the second condition states that if a nondeterministic unstack action fails by dropping the block on the table, in which case the block will be on the table in the next state of the world, then the gripper must be holding the block in the state following that failed state, which can only be satisfied by picking up that block from the table.

Once a failure-recovery strategy is encoded for all four actions in the domain, the original TLPlan search-control formula χ must be modified as follows:

$$\chi' : \Box[\chi \wedge (\chi_{recover}^{unstack} \vee \chi_{recover}^{stack} \vee \chi_{recover}^{pickup} \vee \chi_{recover}^{putdown})],$$

where each $\chi_{recover}^{a}$ is the failure-recovery formula written for each action a in the domain.

3.4.2 ND-TALplanner

Figure 3.8 shows the pseudocode of the ND-TALplanner algorithm. As in TALplanner [KD01], the SearchControl and Progress subroutines are responsible from checking the non-modal control rules and progressing the modal formulas, respectively.

```
Procedure ND-TALplanner(OPEN, N_G, π, solved)
 if OPEN = ∅ then return(π)
 select a pair (N_s, N_χ) ∈ OPEN and remove it
 if N_s satisfies N_G then solved ← solved ∪ {N_s}
 else if N_s ∉ S_π
  actions ← {a | a is a ground instance of an operator in N_op, and
                 a is applicable in s}
  if actions = ∅ then return(FAILURE)
  nondeterministically choose a ∈ actions
  N_χ' ← Progress(N_s, a, N_χ)
  if SearchControl(N_s, N_χ') = FALSE
   then return(FAILURE)
  π' ← append(N_s, a, π)
  π ← π'
  OPEN' ← OPEN ∪ {(N_s', N_χ') | N_s' ∈ γ(N_s, a)}
 else if N_s does not have a π-descendant in (StatesOf(OPEN) ∪ solved) \ S_π then
  return(FAILURE)
 return(ND-TALplanner(OPEN', G, π, solved))
```

Figure 3.8: ND-TALplanner, the nondeterminization of TALplanner for finding strong-cyclic solutions for nondeterministic planning problems. Initially, $OPEN$ is the set of pairs of the form $(\mathcal{N}_s, \mathcal{N}_\chi)$, where $\mathcal{N}_s$ is the TAL formula describing an initial state and $\mathcal{N}_\chi$ is the TAL formula that encodes the search-control information χ. The underlines indicate how the coding from TALplanner is embedded in ND-TALplanner.

A TAL formula $\mathcal{N}_\chi$ developed originally for controlling search in TALplanner can be used/modified to work in ND-TALplanner for nondeterministic planning problems in the same way as described above for the nondeterminization of TLPlan.

3.4.3 ND-SHOP2

Figure 3.9 shows the nondeterminization of SHOP2, called ND-SHOP2. In SHOP2, and therefore in ND-SHOP2, the search-control information for a planning domain is encoded by using *Hierarchical Task Networks (HTNs)* and its progression mechanism is based on decomposing the tasks in those task networks. Consequently, the pseu-

```
Procedure ND-SHOP2(OPEN, G, π, solved)
 if OPEN = ∅ then return(π)
 select a pair (s, w, M) ∈ OPEN and remove it from OPEN
 if s ∈ G then solved ← solved ∪ {s}
 else if s ∉ S_π
  T ← {t | t ∈ w and t has no predecessors}
  nondeterministically choose a t ∈ T
  if t is a primitive task then
   actions ← {(a, σ) | a is an action, σ is a substitution s.t.
                       head(a) = σ(t), and a is applicable in s}
   if actions = ∅ then return FAILURE
   choose (a, σ) ∈ actions
   w' ← σ(w − {t})
   π' ← append(s, a, π)
   π ← π'
   OPEN ← OPEN ∪ {(s', w', M) | s' ∈ ApplyOperator(s, w, t, a, σ, γ)}
  else
   methods ← {(m, σ) | m is an instance of a method in M, σ is a
                substitution s.t. head(m) = σ(t), and m is applicable in s}
   if methods = ∅ then return FAILURE
   choose (m, σ) ∈ methods
   w' ← ApplyMethod(s, w, t, m, σ)
   OPEN' ← OPEN ∪ {(s, w', M)}
 else if s does not have a π-descendant in (StatesOf(OPEN) ∪ solved) \ S_π then
  return(FAILURE)
 return(ND-SHOP2(OPEN', G, π, solved))
```

Figure 3.9: ND-SHOP2, the nondeterminization of SHOP2. In the initial call, $OPEN$ is the set of pairs of the form (s, w, M) where s is an initial state, w is the initial task network, and M is the set of available HTN methods. The underlines indicate the code inherited from the SHOP2 planning algorithm.

docode of the ND-SHOP2 algorithm looks different than that of both ND-TLPlan and ND-TALplanner. However, it is basically a forward search algorithm over a space that is defined by an initial state(s) of the world and the set of actions available to the planner. Thus, it is an instance of the abstract ND-FCP planning procedure.

ND-SHOP2 takes as input a set of goal states G, the empty policy π, the empty set

$solved$, and the $OPEN$ set, which is initially $\{(s, w, M) \mid s \in S_0\}$ where S_0 is the set of initial states, w is the initial task network, and M is the set of available task-decomposition methods.

At each invocation, ND-SHOP2 first selects a tuple (s, w, M) from the $OPEN$ set and removes it. Like its predecessor SHOP2, ND-SHOP2 recursively decomposes the tasks in w into smaller and smaller tasks, until a primitive task (i.e., an action a) is generated for the current state s. Then, it applies a to s and generates the successor states $\gamma(s, a)$. This produces the successor task network w' to be accomplished in each state $s' \in \gamma(s, a)$. The planning process proceeds with each such successor (s', w') until there is no tasks left to be accomplished.

Like in SHOP2, the search-control information in ND-SHOP2 is defined by the current task network that the planner is trying to accomplish at a particular stage of the planning process and the set of HTN methods available to the planner. The acceptable and the progress functions in ND-SHOP2 are both defined by the task-decomposition mechanism that the planner uses. More specifically, in a state s, acceptable$(s, a, \chi = (w, M))$ holds for each action a that is generated by successively decomposing the tasks in the current task network w until we reach task network w' in which a is a primitive task that has no predecessors. Then, progress(s, a, χ) is the successor search-control information (w'', M) where w'' task network generated by removing a from w'. Figure 3.9 also shows the pseudocodes for acceptable and progress. The subroutines ApplyOperator and ApplyMethod are as defined in SHOP2 [NAI+03].

The search-control information for ND-SHOP2 in a nondeterministic planning domain can be produced by modifying the task-decomposition methods that were originally

```
(:method (move-block ?solved)
 ;; method for moving x from y to z
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
       (goal (on ?x ?z)) (different ?x ?z) (clear ?z) (not (need-to-move ?z)))
 ((!unstack ?x ?y) (check-unstack-and-continue-with-stack ?x ?y ?z ?solved))

 ;; method for moving x from y to table
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
       (goal (on-table ?x)))
 ((!unstack ?x ?y) (check-unstack-and-continue-with-putdown ?x ?y ?solved))

 ;; method for moving x from table to y
 (:first (arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on-table ?x)
       (goal (on ?x ?y)) (clear ?y) (not (need-to-move ?y)))
 ((!pickup ?x) (check-pickup-and-continue-with-stack ?x ?y ?solved))

 ;; method for moving x out of the way
 ((arm-empty) (clear ?x) (eval (not (member '?x '?solved))) (on ?x ?y)
  (need-to-move ?x))
 ((!unstack ?x ?y) (check-unstack-and-continue-with-putdown ?x ?y ?solved))

 ;; if nothing else matches, then we're done
 nil
 nil)
```

Figure 3.10: The task-decomposition method that describes the search-control information for ND-SHOP2 in the nondeterministic Blocks World domain.

designed for the classical version of that domain. As an example, consider the nondeterministic versions of Blocks World problems as described in Section 3.1, where an action may fail and drop the block on the table. Here, the task-decomposition method given in Figure 2.3 can be used with a slight modification in order to encode a failure-recovery strategy that tells the planner that, when a block is dropped on the table due to a failure, it needs to pick up that block immediately. To do so, we insert a failure-recovery task after each action in the task-decomposition method of Figure 2.3, and develop a new task-decomposition method for that failure-recovery task.

```
(:method (check-unstack-and-continue-with-stack ?x ?y ?z ?solved)
 ;; If the intended effect of the unstack action occurs, then continue with stack
 ((holding ?x))
 ((!stack ?x ?z) (check-stack-and-continue ?x ?z ?solved))

 ;; If the unstack action fails, then immediately pick-up the block and continue
 ((on-table ?x))
 ((!pickup ?x) (check-pickup-and-continue-with-stack ?x ?z ?solved)))
```

Figure 3.11: The task-decomposition method for the failure-recovery task for the unstack primitive task in the nondeterministic Blocks World domain.

Figure 3.10 shows the modified task-decomposition method for the move-block task. The following describes the task-decomposition method for the failure-recovery task check-unstack-and-continue-with-stack for the unstack primitive task; the methods for failure-recovery tasks for the other actions is very similar. The method for the task check-unstack-and-continue-with-stack specify two different ways to accomplish that recovery task, as shown in Figure 3.11. The first is when the unstack action succeeds; i.e., when its intended outcome of holding the block occurs in the world. In this case, task-decomposition method tells the planner to stack the block to its destination position, as in the original domain description written for SHOP2. The second case is where the action fails and the block is on the table. Then, the method for the failure-recovery task tells the planner to pick that block up immediately and then stack it to its destination.

3.4.4 ND-HSP

HSP, a heuristic-search planner, is a variation of the well-known A^* search algorithm, which is a backtracking forward search algorithm. For this reason, the pseudocode for the nondeterminized version of HSP, called ND-HSP, is the same as the abstract

ND-FCP procedure shown in Figure 3.6.

Although the search-control functions in the original HSP planning algorithm can be used in ND-HSP, this information alone may not help much in pruning the search space in some cases for the following reason. For every state ND-HSP generates during its search, the search-control information (i.e., the distance-cost estimate computed for that state) will specify an action that would be best to reach a goal state from that current state. In the worst case, ND-HSP may explore exponentially many states since it is not possible to use domain-specific information in ND-HSP, in order to encode similar failure-recovery strategies as described above for ND-TLPlan, ND-TALplanner, and ND-SHOP2.

However, in most planning problems, a slight modification can be made to a search-control function for HSP in order to make it work in nondeterministic settings. More specifically, the search-control function acceptable computes the same distance/cost estimates for a state s as HSP, if s is generated by the intended outcome of an action. If s is a failed state, then the modified search-control function returns a heuristic value that will force the planner to choose an action a such that each outcome of a is a state that has been visited before in the current search trace. This modified search-control function works correctly, and it enables us to use in ND-HSP similar kinds of failure-recovery strategies described above.

3.5 Formal Properties of the Nondeterminization Technique

This section presents the formal properties of the nondeterminization method described in the previous sections. The proofs of the theoretical results can be found in Appendix A.1.

The following definitions will be helpful for a clear exposition of the theoretical results. Recall that a planning problem P is *solvable* if there is a solution for it. A planning problem P is χ-*solvable* if there is a solution for it given the search-control information χ. Such a solution will be denoted by π_χ throughout this section. Intuitively, if π_χ is a solution for a planning problem, then the search-control information χ does not prune any actions in π_χ. Note that if a planning problem is χ-solvable then it is solvable.

Let

- Σ be a classical planning domain and Σ' be a nondeterministic version of Σ;
- P be a classical planning problem in Σ and P' be a planning problem in Σ' that is a nondeterministic version of P;
- χ and χ' be the search-control information for Σ and Σ', respectively; and
- Λ be an instance of FCP and ND-Λ be the corresponding instance of ND-FCP.

The following theorems establish that our nondeterminization technique is correct.

Theorem 1 *Suppose one of the search traces of ND-Λ returns a policy $\pi_{\chi'}$ for P' given χ'. Then $\pi_{\chi'}$ is a solution policy for P'..*

Theorem 2 *Suppose that $P' = (S_0, G, \Sigma)$ is χ'-solvable. Then, at least one of the search traces of ND-Λ returns a solution policy.*

The following theorem establishes an upper bound on the time complexity of a nondeterminized planning algorithm for finding solutions in *strongly-connected* planning domains. A planning domain is *strongly-connected* if and only if every state is reachable from any other state in that domain. Such domains are not hard to find. Most well-known classical planning domains are strongly connected (some examples from previous planning competitions [Bac01, FL02] include Blocks-World, Logistics, DriverLog, Zeno-Travel, Depot, and Rover). Note that any nondeterminization of such a domain will also be strongly connected.

The following lemma will be helpful for the main complexity theorems in the nondeterminization technique.

Lemma 3 *Suppose Λ returns a solution plan π for the classical planning problem P. Then, one of the search traces of ND-Λ also returns π for P.*

Then, we get the following theorem:

Theorem 4 *Suppose Λ finds solution plans in time $O(\rho(|\pi_\chi|))$ in a strongly-connected classical planning domain, given the search-control information χ. $|\pi_\chi|$ is the size of the solution plan and ρ is a monotonic function.*

Then ND-Λ finds solutions in time $O(\rho(|\Sigma_{\pi'_{\chi'}}|))$ in a nondeterminized version of that planning domain, where $|\Sigma_{\pi'_{\chi'}}|$ is the size of execution structure for the solution policy $\pi'_{\chi'}$ returned by ND-Λ.

Intuitively, Theorem 4 says that the time complexities of the nondeterminized algorithms are bounded by those of the original algorithms. As a special case, if the original algorithms generate solution plans for planning problems in a strongly-connected classical

domain in polynomial times, then the nondeterminized algorithms also generate solution policies for the nondeterministic versions of those problems in polynomial times. For example, TLPlan, TALplanner, and SHOP2 solve Blocks World problems in polynomial times. Thus, the nondeterminized algorithms ND-TLPlan, ND-TALplanner, and ND-SHOP2 solve the planning problems in the nondeterministic version of Block World that we described in Section 3.1 also in polynomial times.

A corollary immediately follows:

Corollary 5 *Under the conditions of Theorem 4, if the number of possible successors of each state is bounded by a constant, then ND-Λ finds solutions in time $O(\rho(|\pi'_{\chi'}|))$, where $|\pi'_{\chi'}|$ is the size of the solution policy.*

The following complexity result holds in planning domains that are not strongly connected:

Theorem 6 *Suppose Λ finds solution plans in time $O(\rho(|\pi_\chi|))$ in a classical planning domain, given the search-control information χ. $|\pi_\chi|$ is the size of the solution plan and ρ is a monotonic function.*

Then, ND-Λ finds solutions in average time $O(\rho(n) + \frac{bdn}{t}))$, where $n = |\Sigma_{\pi'_{\chi'}}|$ is the size of the execution structure for the solution policy $\pi'_{\chi'}$ returned by ND-Λ given the search-control information χ', b is the maximum number of state-action pairs that are added to any policy after ND-Λ generates a dead-end state-action pair, t is the maximum number of actions applicable to a state, and in every state s, $0 \leq d \leq t$ is the maximum number of actions applicable to s that lead to a dead-end state.

Note that, in planning domains that are not strongly connected, a partial policy generated at any step of our nondeterminized algorithms may induce cyclic executions that have no possibility of reaching the goals, when that policy is executed. In such cases, the nondeterminized algorithms backtrack and try alternative policies as soon as they detect an unacceptable cycle in the current partial policy. However, by the definition of π-descendancy, a planner may detect an unacceptable cycle in the current partial policy only after it performs some additional work after the point that cycle is first introduced in the partial policy. When the planner detects an unacceptable cycle, it backtracks to the point when that cycle is introduced and try alternative policies. In doing so, all of the additional work performed on the current partial policy is lost; hence the additional term in the time complexities of the nondeterminized planners in the theorem above.

Similar as in the strongly-connected case, we have the following corollary:

Corollary 7 *Under the conditions of Theorem 6, if the number of possible successors of each state is bounded by a constant, then* ND-Λ *finds solutions in average time* $O(\rho(|\pi'_{\chi'}|) + \frac{bd|\pi'_{\chi'}|}{t}))$*, where* $|\pi'_{\chi'}|$ *is the size of the solution.*

Note that, although the complexity results reported in Theorems 4 and 6, and their corollaries are true upper bounds, we can show that there exists a much tighter upper bound for weak planning using ND-FCP. This is because weak planning is only a variation of classical (i.e., deterministic) planning in which, in effect, a planning algorithm solves a series of classical planning problems, each of which corresponds to an initial state of the input nondeterministic planning problem. Thus, the above complexity results reduce to the following corrollary:

Corollary 8 *Suppose Λ finds solution plans in time $O(\rho(|\pi|))$ in a classical planning domain, where $|\pi|$ is the size of the solution plan and ρ is a monotonic function. Then, ND-Λ returns weak solutions for nondeterminized versions of those planning problems in time $O(\rho(|\pi|))$.*

We will now describe a set of conditions under which we can guarantee to have an upper bound on the sizes of the policies returned by our nondeterminized planning algorithms. In particular, we describe an upper bound on the sizes of policies returned by these planners in *failure-recoverable* planning domains.

We formalize this notion as follows. Let a, a' be two actions in a nondeterministic planning domain Σ'. Let s be a state in which a is applicable (i.e., $\gamma(s, a) \neq \emptyset$), and let $s' \in \gamma(s, a)$ be an unintended outcome of applying a in s. Let a' be an action that is applicable in s'. Then, a' is a *recovery* action for a if $\gamma(s', a') \subseteq \{s, s', s_i\}$, where s_i is the intended outcome of applying a in s. A nondeterministic planning domain is *failure recoverable*, if for every action there is a recovery action in that domain. It is important to note that most of the planning domains such as Blocks-World, Logistics, Depot, and ZenoTravel and others are failure recoverable planning domains.

If a classical planning algorithm Λ finds a solution plan π in a classical failure-recoverable planning domain using the search-control information χ, then ND-Λ also returns solutions of size $O(|\pi|)$ for a nondeterminized version of that planning domain using a search-control information χ' for that domain, where $|\pi|$ is the size of the solution plan. This is because failure-recoverable planning domains do not require any extra planning effort for the action failures; thus, the nondeterminized algorithms are guaranteed to

run in polynomial times, if their deterministic counterparts do so.

3.6 Experimental Evaluation

This section presents an experimental evaluation of one of the nondeterminized planning algorithms, namely ND-SHOP2, the strong-cyclic nondeterminization of SHOP2 shown in Figure 3.9. The experimental evaluation compares ND-SHOP2's performance and scalability with MBP [BCP+01], the best previous planning system for nondeterministic domains.

MBP is a model-checking based planner that implements the weak, strong, and strong-cyclic algorithms described in Section 2.3.2. MBP is a general planning system that is composed of two stages. In the first stage, the input planning domain and the problems, which are expressed in the high-level action language $\mathcal{AR}$ [CGGT97], are compiled into Binary Decision Diagrams (BDDs). In the second stage, different planning algorithms are applied to the specified planning problems and domains as described in [CPRT03]. The MBP planning system is written in C++ [BCP+01] and it uses the *Colorado University Decision Diagram (CUDD)* package[3] for an implementation of Binary Decision Diagrams.

I implemented ND-SHOP2 in LISP. The reason for the LISP implementation of the planner is that SHOP2 was implemented in LISP and the implementation of ND-SHOP2 builds on SHOP2's source code, as the former being a generalization of the latter. The experimental results do not include the compilation times for ND-SHOP2 and MBP; in-

[3]CUDD is accessible via http://vlsi.colorado.edu/~fabio/CUDD/

stead, the source codes for the planning algorithms were compiled before the experiments began. Following [PBT01], the experimental results included the times that took for MBP to preprocess its input planning problems to convert them into BDDs — this had very little impact on the results reported in the subsequent sections as the preprocessing times were always in the order of a few seconds in all experimental problems.

All of the experiments described in the subsequent sections were performed on an AMD Duron 900Mhz laptop computer with 256MB memory running Fedora Core 2 Linux. The experiments involved three planning domains: two nondeterministic versions of the classical Blocks World domain and the Robot-Navigation domain that was used in [PBT01, CPRT03]. For each domain, both ND-SHOP2 and MBP were provided as input the same planning operator descriptions (i.e., action desciptions) in their respective representational languages. All of the domain descriptions, problem descriptions, and the random problem generators used in these experiments will be accessible via http://www.cs.umd.edu/users/ukuter/nfcp/. The subsequent sections describe the experimental planning domains, problems, and results, as well as a complexity analysis of ND-SHOP2 on these domains.

3.6.1 Nondeterministic Blocks World

The first nondeterministic version of Blocks World used in this experimental evaluation was as follows. Each action may have two kinds of outcomes: (1) its intended effect (the same effect as the action would have in the original Blocks World domain), and (2) a failed effect such that the action may fail to change the state of the world at all – e.g., a

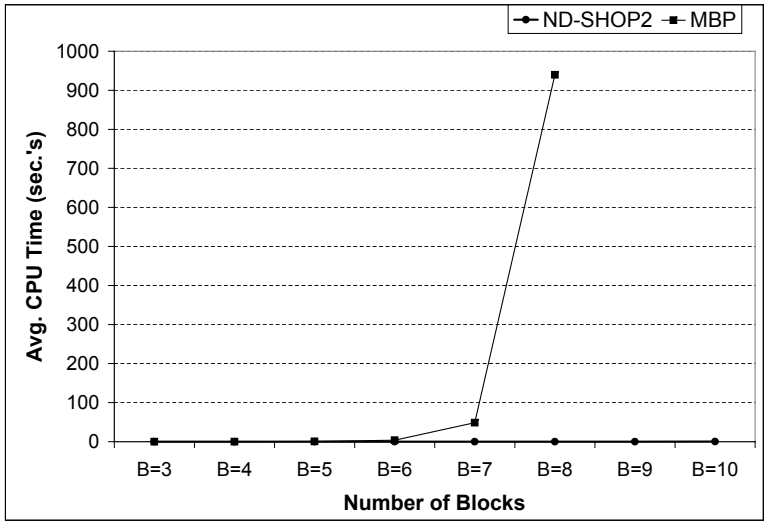

Figure 3.12: Average running times of ND-SHOP2 and MBP in the first version of the nondeterministic Blocks World domain, as a function of the number of blocks.

pickup operator may fail to pick up a block from the table or a stack operator may still be holding the block it intended to stack on another one.

Figure 3.12 shows the results of the experiments in the nondeterministic Blocks World domain. Each data point is the average of 20 random problems. For MBP, there are no data points for $n > 8$ because it was unable to solve any problems within the alloted time (30 minutes per problem). These results show that MBP is extremely sensitive to the size of the problems, which is the number of blocks in this case. On the other hand, the performance of SHOP2 seems to be not affected by the increasing size of the problems. In particular, the time required by MBP grows exponentially with the increasing size of the problems (the logarithm of MBP's CPU time is linear), whereas curve fitting on ND-SHOP2's running time shows that it grows only polynomially.

The reason for the polynomial behavior of ND-SHOP2 on these problems are as follows. The nondeterminism in these problems are due to the failed effects of the actions we described above; an action may fail to change the state of the world. This means

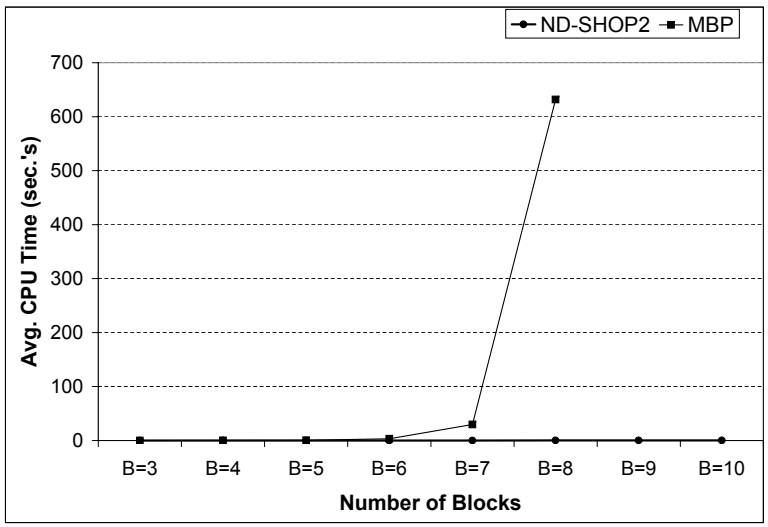

Figure 3.13: Average running times of ND-SHOP2 and MBP in the second version of the nondeterministic Blocks World domain, as a function of the number of blocks.

that the failure of an action a in a state s can only take the planner back to s itself. This will induce a cycle in the search space of ND-SHOP2, but this cycle is a valid cycle according to the definition of strong-cyclic solutions. For this reason, we can use the same search-control information of SHOP2 developed for Blocks World in ND-SHOP2 for this nondeterministic version. Since this search-control information enables SHOP2 to find solutions in polynomial times, so does ND-SHOP2.

The second nondeterministic version of the Blocks World domain is the one described in Section 3.1. To summarize, in this nondeterministic version, the actions may have three possible outcomes: the same two outcomes as the one above and a third outcome such that an action may fail by dropping the block onto the table. Note that, unlike before, this kind of failure in the actions may possibly produce new states that the planner needs to explore to generate strong-cyclic solutions to the planning problems.

Figure 3.13 shows the results of the experiments on the second nondeterministic

version of Blocks World. Like before, each data point is the average of 20 random problems. For MBP, there are no data points for $n > 8$ again because it was unable to solve any problems within the alloted time (30 minutes per problem). The logarithm of MBP's CPU time is also linear; thus on this problem domain, like the other one, MBP takes exponential time.

Curve-fitting on ND-SHOP2's running time shows it growing at only about $\Theta(n^5)$, and a complexity analysis confirms this polynomial behavior (see Section 3.7). The reason for ND-SHOP2's polynomial behavior is that it uses a search-control strategy that involves task-decomposition methods for recovering from the action failures, as described in Section 3.4. In particular, each time an action drops the block on the table, ND-SHOP2 picks the block up immediately, and tries that action again. As a result, ND-SHOP2 is guaranteed to produce a policy of size $O(n)$ where n is the number of blocks, as its predecessor SHOP2 would do in the original Blocks World domain. In contrast, MBP produces exponential-size policies that tell what to do in most of the states of the world.

3.6.2 Robot Navigation

The third experimental domain in the evaluation of ND-SHOP2 was the Robot Navigation domain that was used as a benchmark domain for MBP in [PBT01, CPRT03]. This domain is a variant of a similar domain described in [KBSD97]. It consists of a building with 8 rooms connected by 7 doors. In the building, there is a robot and there are a number of packages in various rooms. The robot is responsible for delivering packages from their initial locations to their final locations by opening and closing doors, moving

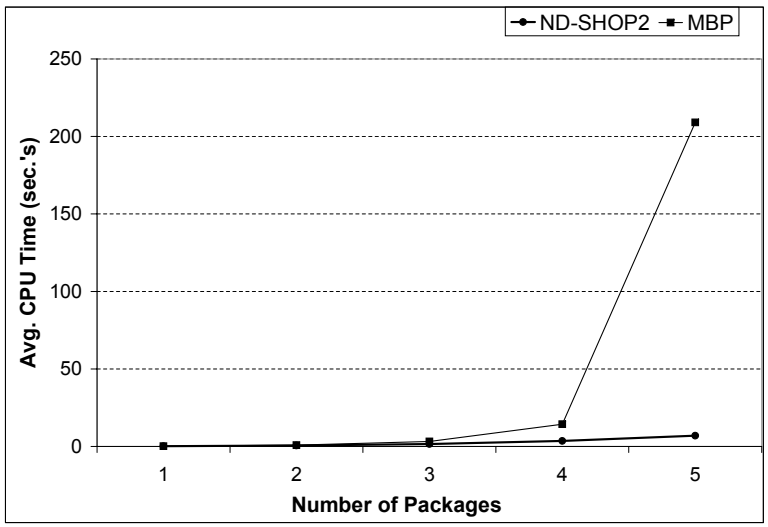

Figure 3.14: The average running times of ND-SHOP2 and MBP on Robot-Navigation problems as a function of the number of packages, when the number of kid-doors in the domain is fixed to 7.

between rooms, and picking up and putting down the packages. The robot can hold at most one package at any time. Nondeterminism is introduced via a "kid" that can close any of the open doors that are designated initially as "kid-doors."

The experiments in this domain compared ND-SHOP2 and MBP with the same set of experimental parameters as in [PBT01]: the number of packages n is ranged from 1 to 5, and the number of kid-doors k is ranged from 0 to 7. Figure 3.14 shows only the results for $k = 7$ and $n = 1, \ldots, 5$; this illustrates the behavior of the algorithms as the sizes of the problems increase. As in [PT01], the CPU time for MBP's includes both its preprocessing and search times. Omitting the preprocessing times would not have significantly affected the results: they were never more than a few seconds, and usually below one second.

These results confirm the ones in [PBT01]: in both their experiments and the ones described here, MBP's CPU time grows exponentially (the logarithm of the data grows

linearly) in the size of the problem. In contrast, the data for ND-SHOP2 show its CPU time growing polynomially (see Section 3.7 for a complexity analysis).

Note that if there are k kid doors in the domain, then there are 2^k possible initial states. However, in the representation language for the Robot Navigation domain, a policy can say things along the lines of "if we are at door number 3 and it is open, then go through it," rather than having to give explicitly all of the exponentially many states of the world in which we're at door number 3 and the door is open. More specifically, ND-SHOP2 does not represent the fact that a kid door may be either open or closed explicitly in a state. Instead, when the robot is in front of a kid door, ND-SHOP2 splits this state in two states such that the door is open in one of them and it is closed in the other, and plans an action for each of such states. After that, since the robot is done with door at this point, it merges these two states again into one in which the information about the openness or closedness of that particular door has been disappeared. Note that the splitting and merging operations performed this way are easily encoded in ND-SHOP2's task-decomposition methods.

Because of this, problems in the robot-navigation domain have strong-cyclic solutions of linear size, and these are the solutions that ND-SHOP2 finds. In fact, this is the main reason for ND-SHOP2's fast performance relative to MBP in this domain. Although MBP represents policies in a similar compact way, it apparently does not exploit this representation well enough to produce policies of polynomial size in its backward search algorithms.

To illustrate the scalability of the algorithms as the amount of nondeterminism increases, Figure 3.15 shows the results for $n = 5$ and $k = 1, \ldots, 7$. In each case, MBP

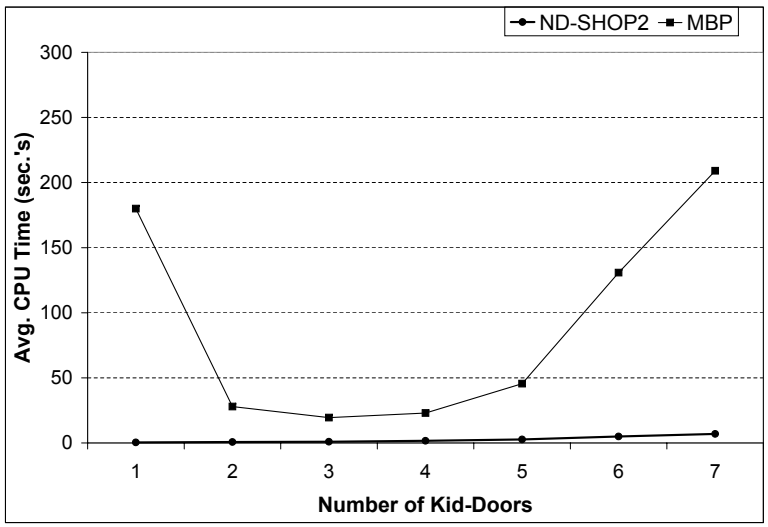

Figure 3.15: The average running times of ND-SHOP2 and MBP on Robot-Navigation problems as a function of the number of kid-doors, when the number of packages in the domain is fixed to 5.

takes one to two orders of magnitude more time than ND-SHOP2. The closest run times occurred at $k = 4$, where MBP required about 15 times as much time as ND-SHOP2 did.

3.7 A Complexity Analysis on the Experimental Results

This section presents the complexity analysis of space and time requirements for ND-SHOP2 on the planning domains used in the experimental evaluation. This analysis is based on the current implementation of ND-SHOP2, as well as on the properties of the task-decomposition methods written for these domains.

Proposition 1 *ND-SHOP2 generates policies of size $O(n)$, where n is the number of objects in Robot Navigation, and the number of blocks in both of the nondeterministic versions of the Blocks World domains, all of which are as described above.*

Proof. Both nondeterministic Blocks World and Robot Navigation admit polynomial-sized solutions. These solutions have the following characteristics:

- In the nondeterministic Blocks World, there are two cases for each block. In the first case, the planner picks up (or unstacks) the block and puts it down (or stacks) it to its goal location. In the second case, the planner cannot move the block to its goal location so it moves it onto the table using the two actions above. Such a block needs to be moved to its goal location later; therefore, the planner performs four actions for that block in this case. However, note that during any of these actions in either case, the planner may drop the block on the table. If the table is not the block's goal location, the search-control strategies above tell the planner to pick the block immediately. Therefore, for each block, a solution includes at most 5 actions to move it from its initial position to its goal. Thus, the size of a solution is $O(n)$, where n is the number of blocks.

- In Robot Navigation, for each object, the search-control strategies mentioned above tell the planner to move the robot from its current position to a location where there is an object, pick up the object, move from that location to the object's goal location, and put it down. Each move operation is a sequence of actions that moves the robot between two rooms that are connected to each other via a door. In the fixed building map in this domain, the maximum distance between two rooms is 5, hence the maximum number of move actions to get the robot from one room to another. Note that nondeterminism via the kid doors does not add any complexity here due to representation we described above: that is, in the representation language for

the Robot Navigation domain, a policy can say things along the lines of "if we are at door number 3 and it is open, then go through it," rather than having to give explicitly all of the exponentially many states of the world in which we're at door number 3 and the door is open. More specifically, when the robot is in front of a kid door, the search-control strategy splits this state in two states such that the door is open in one of them and it is closed in the other, and plans an action for each of such states. After that, since the robot is done with door at this point, it merges these two states again into one in which the information about the openness or closedness of that particular door has been disappeared.

Thus, if a kid door is open, the robot moves through it; otherwise, it opens and moves through it. If the door remains closed after the robot opens it, there is nothing to do for the planner since the planner already generated an action for this case. As a result, for each object, a solution specifies at most 12 actions; therefore, the size of a solution is $O(n)$, where n is the number of objects.

■

Proposition 2 *ND-SHOP2 generates policies in time $O(n^5)$, where n is the number of objects in Robot Navigation, and the number of blocks in the nondeterministic versions Blocks World domains, all of which are as described above.*

Proof. Note that the search-control strategies described in the proof of Proposition 1 enable the planner to generate a solution of $O(n)$ size in $O(n)$ many iterations since, at each iteration, ND-SHOP2 generates one of the actions described above and inserts it

into the current partial policy. The dominant factor in each iteration of ND-SHOP2 is to maintain the data structures we have implemented for checking cycles induced by the partial policies generated by the algorithm — i.e., the π-descendancy in the pseudocode of Figure 3.9.

Before we get into the details of the complexity of cycle checks in our implementation, we need to analyze the size of a state in our domains since our checks are mostly based on it. [GN92] shows that the size of a state in the classical Blocks World domain is $O(n)$ where n is the number of blocks in the domain. Our nondeterminized version of this domain also has this property since the nondeterminization process does not have any effect on the states of the world.

In the Robot Navigation domain. Since, in a state, there is only one atom that describes the location of the robot and there is at most one atom that describe whether a door is open or not, the size of a state in this domain does not influenced by the location of the robot. The size of a state is also not exponential in the status of the kid doors due to our representation of the Robot Navigation as described in the proof of Proposition 1. Furthermore, since the domain is a prescribed map of a floor in a building, the number of atoms that specify connectivity information of the possible rooms is fixed. In fact, the size of the state only changes by the number objects in the domain. More specifically, if there are n objects in the domain, then there are n atoms that describe the locations of these objects. This is true since an object can be only in one room in any state. Therefore, the size of a state in this domain is $O(n)$.

One way to check π-descendancy is to perform a search over the execution structure induced by the partial policies generated during planning. In order to avoid this search,

we took an alternative approach in which we have implemented a data structure, called the *reachable list*. The reachable list keeps the set of states in the open and the $solved$ lists that are reachable from every state in the execution structure for the partial policy generated during planning. In other words, this list holds the π-descendancy information for every state explored by the algorithm until a particular iteration. This data structure enables us to perform a π-descendancy test in polynomial time in the size of the partial policy generated until that iteration, and by Proposition 1, this means that we can perform such checks in $O(n)$ time, where n is the number of blocks in one domain and it is the number of objects in the other.

However, in order to keep the correct π-descendancy information in each iteration, we need to update this list in each iteration when we remove a state from the open list and select an action for it. Let s be such a state in an iteration of the algorithm. Our current implementation performs this update as follows: for every state s' in the partial policy, we first find the set of π-descendants of s' using the reachable list. Then, we find the state that was planned for in this iteration in this set. Finding the set of π-descendants of s' requires $O(n)$ time since the size of the reachable list is the same as the size of the partial policy in this iteration. Finding the particular state in the set of π-descendants of s' requires $O(n^2)$ time since we compare the atoms in s with the atoms of every state in that set. [4] After finding s in that set we remove it and insert the successors of s into that set.

Therefore, updating the reachable list for each of the states in a partial policy requires time $O(n^3)$. Since the number of such states in the partial policy is $O(n)$, the

[4]Note that, due to the search-control strategies described above, the set of π-descendants of a state is always bounded by a constant so we do not include the size of that set in our analysis.

algorithm requires $O(n^4)$ time to perform this update. Since this update needs to be done at each iteration of the algorithm and the number of iterations required to return a solution in our domains is $O(n)$, ND-SHOP2 took $O(n^5)$ time in our experiments. ■

It would also be possible to write nondeterministic versions of the blocks world with more complicated kinds of nondeterminism: for example, an action could drop the block not just onto the table, but onto any clear block. Although this experimental evaluation does not include such cases, the complexity analysis above suggests that such an experimental study would yield results similar to the above. ND-SHOP2 would take polynomial time and space, although the space would this time be quadratic rather than linear (it would immediately pick up the fallen block again, but in the worst case there would be $O(n)$ different places to pick up this block from). MBP would take exponential time in the size of the problem, for the same reasons as before.

Chapter 4

Forward State-Space Splitting in Nondeterministic Domains

The previous chapter has described a way to generalize forward-chaining classical planning algorithms to work in nondeterministic domains. The original planning algorithms are able to use domain-independent or domain-specific search-control information to focus their search in classical planning domains. The nondeterminization technique preserves the ability to use search control for efficient planning in the nondeterministic planning domains, as experimentally demonstrated in Section 3.6 with one of the nondeterminized planning algorithms; namely ND-SHOP2, a generalization of the SHOP2 planner [NAI+03] that can use domain-specific information encoded as *Hierarchical Task Network (HTNs)* for controlling its search.

Despite the success of ND-SHOP2 in the experimental evaluation described in the previous chapter, there are several classes of planning problems in nondeterministic domains for which effective search control may not be available either because of the complexity of the domain. This chapter first describes in Section 4.1 some examples of the planning domains in which this is the case and demonstrates that MBP, without using any search control, performs better than ND-SHOP2 in such cases. Then, Section 4.2 describes a novel planning technique, called *Forward State-Space Splitting* (or FS^3 for short), for planning in nondeterministic domains. The rest of the chapter is dedicated to an extensive discussion on this planning procedure and the theoretical and experimental

analysis of it.

In particular, FS^3 is an abstract planning procedure that is designed to combine the advantages of using search control during planning with that of BDD-based representations of planning problems and domains as in MBP. Instances of FS^3 include FS$^3_{\text{TLPlan}}$ that combines planning with control rules as in ND-TLPlan and ND-TALplanner with BDDs, and FS$^3_{\text{SHOP2}}$ that combines HTNs as in ND-SHOP2 with BDDs. Our experiments with FS$^3_{\text{SHOP2}}$ demonstrated that FS$^3_{\text{SHOP2}}$ was never dominated by either MBP or ND-SHOP2, and could easily deal with problem sizes that neither MBP nor ND-SHOP2 could scale up to. Furthermore, FS$^3_{\text{SHOP2}}$ could solve problems about two or three orders of magnitude faster than MBP and ND-SHOP2.

4.1 ND-FCP vs. Planning with BDDs

The reason that ND-SHOP2 was able to outperform MBP in the experiments of the previous chapter is the effective search-control information provided to the planner to prune the search space. When such information is not available, however, ND-SHOP2 is a simple forward state-space search algorithm that is not to be expected to be effective compared to MBP since the latter uses propositional formulas for a compact representations of sets of states and of transformations over such formulas for efficient exploration in the search space. Thus, it is reasonable to hypothesize that the planning techniques developed using search-control and compact representations perform well on different kinds of planning problems and domains. This section describes two sets of experiments with ND-SHOP2 and MBP in order to verify this hypothesis. One of these experiments

Table 4.1: Comparisons between ND-SHOP2 and MBP on CHAIN problems, with increasing number n of rooms.

$n =$	10	20	30	40	50	60	70	80	90	100
MBP	0.068	0.114	0.157	0.192	0.287	0.389	0.354	0.452	0.583	0.742
ND-SHOP2	0.010	0.040	0.120	0.240	0.430	0.630	1.000	1.260	1.620	2.040

were performed on the toy CHAIN Domain described in [CPRT03] and the other in a pursuit-evasion game, called Hunter-Prey domain, described in [KS95].

All experiments were run on an AMD Duron 900MHz laptop with 256MB memory, running Linux Fedora Core 2. If a planning algorithm failed on a problem (i.e., it ran out of memory or it could not solve the problem within a time limit of 40 minutes), it was run again on another problem of the same size. Each data point on which the planning algorithm failed more than five times is omitted from the experimental results, but those data points where it failed 1 to 5 times are included. Thus the experimental results make the performance of the failed planner look better than it really was compared to the other planner, but this makes little difference since the former performed much worse than the latter in each experimental case the former failed.

In all of these experiments, both ND-SHOP2 and MBP were provided as input the same planning operator descriptions (i.e., action descriptions) for the CHAIN and the Hunter-Prey domains in their respective representational languages.

In the CHAIN domain, there are n rooms, marked as $i = 1, \ldots, n$. The objective is to start from room $i = 1$ and to go to the room $i = n$. Each consecutive room i and $i + 1$ share two doors. The planners do not know which of the doors is open or close between the rooms; thus, the number of possible successors of a state in this version of the domain is exponential in the number of rooms left to visit.

Table 4.2: Comparisons between ND-SHOP2 and MBP on larger CHAIN problems, with increasing number n of rooms. In this table, "$-$" shows the cases where the policy representations in ND-SHOP2 required more memory than that was available.

$n =$	50	100	150	200	250	300
MBP	0.287	0.742	2.874	3.830	7.137	11.097
ND-SHOP2	0.430	2.040	4.370	–	–	–

The experiments with the CHAIN domain compared the running times required to solve planning problems by ND-SHOP2 and MBP, varying the number of rooms in the domain. Tables 4.1 and 4.2 show the results. These results illustrate that ND-SHOP2 was not able to solve large planning problems in this domain, where there are 200 rooms and more, due to memory-overflow errors. On the other hand, MBP was able to solve all of the planning problems in this test suite. The reason for this difference in the behavior of the two planners is that, although ND-SHOP2's search-control information focuses the planner to a particular room being visited, the sizes of the explicit policy representations in ND-SHOP2 become very large; hence the memory overflows to store those policies in the memory. On the other hand, the size of MBP's BDD-based policy representations does not grow with the size of the problems, and therefore, MBP did not have any memory problems in these experiments.

In the Hunter-Prey domain, there is a hunter and a prey in an $n \times n$ grid world. The task of the hunter is to catch the prey in the world. The hunter has five possible actions; namely, north, south, east, west, and catch. The prey has also five actions: it has the same four moves as the hunter, and an action to stay still in the world. The hunter can catch the prey only when the hunter and the prey are at the same location at the same time in the world. In the representation of the Hunter-Prey domain, the prey does not appear as

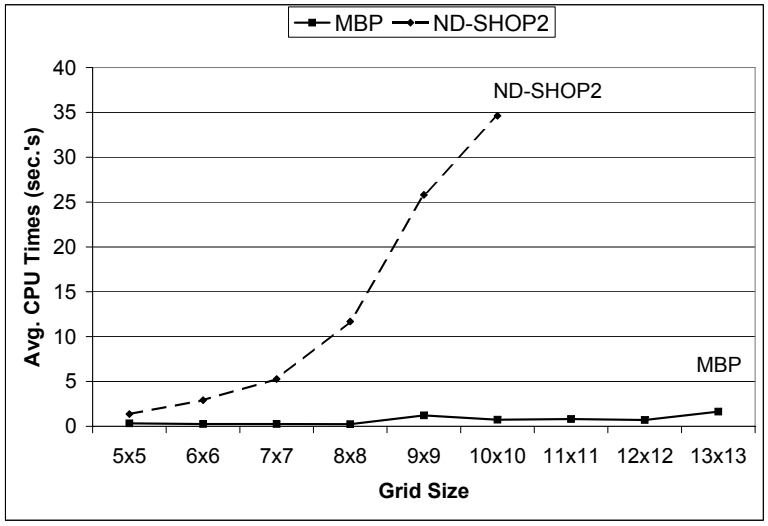

Figure 4.1: Average running times in sec.'s for MBP and ND-SHOP2 in the Hunter-Prey domain as a function of the grid size, with one prey. ND-SHOP2 was not able to solve planning problems in grids larger than 10×10 due to memory-overflow problems.

a separate agent. Instead, the prey's possible actions are encoded as the nondeterministic outcomes for the hunter's actions.

Figure 4.1 shows the average running times required by MBP and ND-SHOP2, as a function of increasing grid sizes. These results are obtained by running the two planners over 20 randomly-generated problems for each grid size, and then, by averaging the results. ND-SHOP2 ran out of memory in the large problems of this domain since (1) the solution policies in this domain are very large to store using an explicit representation, and (2) the search space does not admit a structure that can be exploited by search-control heuristics. Note that this domain allows only for high-level strategies for the hunter such as "look at the prey and move towards it," since the hunter does not know which actions the prey will take at a particular time. MBP, on the other hand, clearly outperforms ND-SHOP2 in these experiments, demonstrating once again the advantage of using BDD-based representations over explicit ones.

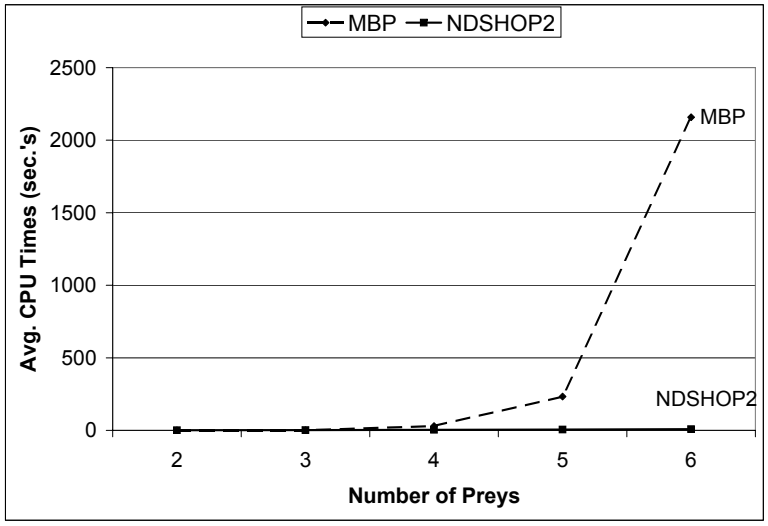

Figure 4.2: Average running times in sec.'s for MBP and ND-SHOP2 in the Hunter-Prey domain as a function of the number of preys, with a fixed 4×4 grid.

Another set of experiments on Hunter-Prey problems focused on a variation of the domain in which there are more than one prey to catch. In this version of the planning domain, the movements of prey are dependent on each other, assuming that a prey cannot move to a location next to another prey in the world. Figure 4.2 shows the results in this adapted domain, with the 4×4 grid world: ND-SHOP2 is able to outperform MBP in this domain. The reason for the difference in these results compared to the previous ones is that this adapted domain allows much more powerful strategies for the hunter: e.g., "choose one prey and chase it while ignoring others; when you catch that prey, choose another and chase it, and continue in this way until all of the prey are caught." ND-SHOP2, using this strategy, is able to avoid the combinatorial explosion due the prey's actions in the world. On the other hand, the BDD-based representations in MBP explode in size since the movements of the preys are dependent to each other, and MBP's backward-chaining breadth-first search techniques apparently cannot compansate for such

an explosion.

The results of the experiments described in this section clearly suggest that planning with search control and with BDD-based state representations are two complementary techniques: in some planning domains the former is exponentially faster than the latter, and in others the reverse is true. The subsequent sections further investigate these two techniques and describe a way to combine them.

4.2 ND-FCP + BDDs = Forward State-Space Splitting (FS^3)

Forward State-Space Splitting (FS^3) is a forward-chaining abstract planning procedure that provides a way to combine the ability of exploiting search-control information in planning with symbolic model-checking techniques in a single planning framework. The symbolic model-checking techniques used in FS^3 are BDDs, as in the MBP planner. FS^3 can exploit search-control information encoded as HTNs as in ND-SHOP2 and control rules as in ND-TLPlan or ND-TALplanner.

Figure 4.3 shows the FS^3 planning procedure for generating solutions in nondeterministic planning domains. The input for the planning procedure FS^3 include the set G of goal states and the empty policy π. The $OPEN$ set is the set of pairs of the form (S, χ) where S is a set of states and χ is the search-control information to be used in all of the states in S.

In any invocation, FS^3 requires the search-control information χ be ground — i.e., χ contains no variable symbols in its representation. This assumption is due to the need that FS^3 evaluates χ in a BDD representation of a set of states, which is based on propo-

Procedure FS$^3(OPEN, G, \pi)$
$OPEN \leftarrow \{(S \setminus (G \cup S_\pi), \chi) \mid (S, \chi) \in OPEN$ and $S \setminus (G \cup S_\pi) \neq \emptyset\}$
if NoGood$(\pi,$ StatesOf$(OPEN), G, S_0)$ then return(FAILURE)
if $OPEN = \emptyset$ then return(π)
select a situation (S, χ) from $OPEN$ and remove it
$F \leftarrow \{(S \cap S_a, a,$ progress$(S \cap S_a, a, \chi)) \mid$ acceptable$(S \cap S_a, a, \chi)$ holds$\}$
if $F = \emptyset$ then return(FAILURE)
nondeterministically choose $F' \subseteq F$
$S_\cup \leftarrow \bigcup_{(S \cap S_a, a, \chi') \in F'} (S \cap S_a)$
if $S_\cup \neq S$ then return(FAILURE)
$S_\cap \leftarrow \bigcup_{(S', a', \chi') \in F', (S'', a'', \chi'') \in F'} (S' \cap S'')$
if $S_\cap \neq \emptyset$ then return(FAILURE)
$OPEN' \leftarrow$ Compute-Successors$(F', OPEN)$
$\pi' \leftarrow \pi \cup \{(s, a) \mid (S \cap S_a, a, \chi') \in F'$ and $s \in S \cap S_a\}$
return(FS$^3(OPEN', G, \pi')$)

Figure 4.3: FS^3, an abstract planning procedure that use search-control information to focus the search for generating solutions in nondeterministic domains. In the initial call of the procedure, π is the empty policy and $OPEN$ is the set that contains the pair (S_0, χ) where S_0 is the set of initial states and χ_0 is the initial search-control information.

sitional formulas (i.e., logical formulas over ground atoms).[1]

For the purposes of clarity of the discussion in the rest of this chapter, we will call a pair (S, χ) as a *situation*. Initially, $OPEN$ contains only the initial situation (S_0, χ_0), where S_0 is the set of initial states and χ_0 is the initial search-control information. Starting with the initial situation, FS^3 recursively generates successive sets of situations until a solution for the input planning problem is generated. At each iteration of the planning process, FS^3 first checks the $OPEN$ set of situations for cycles and goal states: for every situation $(S, \chi) \in OPEN$, the algorithm removes any state s from S that either appears

[1] In the general case where the search-control information contains variable symbols, FS^3 can be extended by a preprocessing phase that creates possible ground instances automatically. However, the current implementation of the planning procedure does not perform this phase; this is one of the near-future works planned to extend FS^3.

already in the policy, i.e., $s \in S_\pi$, or that appears in the set of goal states G. In the former case, an action has already been planned for s, and in the latter case, no action should be planned for s. During this operation, if the set S of states in a situation becomes empty, then FS^3 simply discards that situation since there is no further exploration it can perform from this situation on.

After processing the cyclic and goal states in the $OPEN$ situations, FS^3 perform a correctness test on the $OPEN$ situations and the current partial policy π. This test involves verifying that the current partial policy π is a candidate solution for the input planning problem. In Figure 4.3, the NoGood function is responsible for this test. The formal definition of this subroutine depends on whether the FS^3 planning procedure is used for generating weak, strong, or strong-cyclic solution for the input planning problem, and it is described in the following section. NoGood takes as input the states of the $OPEN$ set, which is computed by the function StatesOf:

$$\textsf{StatesOf}(OPEN) = \{s \mid (S, \chi) \in OPEN \text{ and } s \in S\}$$

If the current partial policy π is not a candidate solution, then FS^3 returns from the current search trace by FAILURE. Otherwise, if there are no open situations to be explored further (i.e, $OPEN = \emptyset$), then π is a solution to the underlying planning problem. This is true since π does not violate the requirements of the input problem, as it passed all of the NoGood tests from the start of the planning process to this point.

Suppose there is an $OPEN$ situation (S, χ) in an invocation of FS^3. Then FS^3 generates (1) an action a for each state s in S given the search-control information χ and (2) the successor search-control information χ' to be used in the states that arise from

Procedure Compute-Successors$(F, OPEN)$
$OPEN' \leftarrow OPEN \cup \{(succ(S,a),\chi) \mid (S,a,\chi) \in F\}$
$OPEN' \leftarrow \{(\textsf{Compose}(\chi, OPEN'),\chi) \mid (S,\chi) \in OPEN'\}$
return $OPEN'$

Figure 4.4: The Compute-Successors procedure.

applying a in s. More specifically, FS^3 computes a set F tuples of the form $(S \cap S_a, a, \chi')$, where $S \cap S_a$ is the subset of states in S in which the action a is applicable, and it is acceptable to apply a in those states given the current search-control information χ. The search-control information to be used in any state that is generated by applying a in a state in $S \cap S_a$ is $\chi' = \textsf{progress}(S \cap S_a, a, \chi)$.

If F is the empty set then this means that there is a state in S for which there is no action given the current search-control information χ. In this case, FS^3 returns FAILURE. Otherwise, FS^3 nondeterministically chooses a subset F' of F. If the subset F' specifies one and only one action for every state in S (i.e., $S = \bigcup_{(S',a,\chi') \in F'} S'$ and there are no two tuples, say (S', a', χ') and (S'', a'', χ'), in F' such that $a' \neq a''$ and $S' \cap S'' \neq \emptyset$), the algorithm computes the set of all successor situations that can be generated by applying those actions in the states of S. Otherwise, FS^3 returns FAILURE.

The Compute-Successors subroutine generates the new $OPEN'$ set of situations to be explored in the next iterations of the planning process. The formal definition of this subroutine is shown in Figure 4.4. For each tuple $(S, a, \chi) \in F$, Compute-Successors first generates the set of states that arises from applying a in S by using the function

$$succ(S,a) = \{s' \mid s \in S \text{ and } s' \in \gamma(s,a)\}.$$

The next situation corresponding this action application is defined as $(succ(S,a),\chi)$.

Once Compute-Successors generates the all of the next situations be explored, it composes the newly-generated situations with respect to their search-control information. More formally, the Compose function of Figure 4.4 is defined as follows:

$$\textsf{Compose}(\chi, OPEN) = \{s \mid (S, \chi) \in OPEN \text{ and } s \in S\}.$$

The composition of a set of situations is an optimization step in the planning process. The progression of open situations may create a set of situations in which more than one situation may specify the same search-control information. Composing such situations is not required for correctness, but it has the advantage of planning with more compact representations.

4.3 Weak, Strong, and Strong-Cyclic Planning with FS^3

The abstract planning procedure FS^3 can be used for weak, strong, and strong-cyclic planning by using different NoGood subroutines, each of which specifies the different conditions required for a policy to be a weak, strong or strong-cyclic solution for a planning problem. This section presents the definitions for these routines.

Weak Planning. The NoGood function for weak planning is mainly responsible for checking if there is an acyclic path from each initial state to a goal state in π. The formal definition of this function is as follows:

```
Procedure NoGood(π, S_OPEN, G, S_0) /* for Weak Planning */
  S'' ← ∅; S ← G ∪ S_OPEN
  while S'' ≠ S
    S'' ← S
    S' ← {s' | (s', a) ∈ π and γ(s', a) ∩ S ≠ ∅}
    π ← π \ {(s, a) | s ∈ S and (s, a) ∈ π}
    S ← S ∪ S'
  if S_0 ⊆ S then return FALSE
  return TRUE
```

This computation involves a backward search starting from the goal states and the $OPEN$ states, the set S in the pseudocode for NoGood above, toward the initial states S_0 of the planning problem input to FS^3. The $OPEN$ states are the ones in the situations in the $OPEN$ set. The backward search is based on the WeakPreimage operation described in Section 2.3.2. The search stops when all states that can be reached from the goal and the $OPEN$ states by the WeakPreimage operations are generated in S. At this point, S contains all the states from which there is an acyclic path to a goal state. Then, NoGood simply checks if the initial states are in S. If so, the input partial policy π does not violate the requirements of weak planning, so it returns FALSE (which tells FS^3 that π is actually good). Otherwise, it returns TRUE, forcing FS^3 to backtrack.

Strong Planning. In strong planning, a policy must induce an execution trace to a goal state from every state that is reachable from the initial states and there should be no cycles in the execution structure induced by that policy. This can be checked as follows:

```
Procedure NoGood(π, S_OPEN, G, S_0) /* for Strong Planning */
  S' ← ∅; S ← G ∪ S_OPEN
  while S' ≠ S
    S' ← S
    S ← S ∪ {s' | (s', a) ∈ π, and γ(s', a) ⊆ S}
    π ← π \ {(s, a) | s ∈ S and (s, a) ∈ π}
  if S_0 ⊆ S and π = ∅ then return FALSE
  return TRUE
```

Note that the above NoGood function for strong planning have several similarities with the one for weak planning in that both are backward search procedures that start from the goal and the $OPEN$ states and perform a search towards the initial states. NoGood for strong planning, however, uses the StrongPreimage function described in Section 2.3.2, rather than the WeakPreimage function. At each iteration, the NoGood for strong planning removes those state-action pairs from π that have been verified to be in the StrongPreimage of the goal states. At the end of the backward search, if there is a state-action pair left in the policy, then it means that the policy induces a cycle in the execution structure, and therefore, it can not be a strong solution for a planning problem. In this case, NoGood returns TRUE, meaning that the input partial policy π is not good and violates the requirements of being a strong solution.

Strong-Cyclic Planning. The definition for the NoGood function for strong-cyclic planning is very similar to that for weak planning, except that it ensures that every state-action pair in the input partial policy π is processed by the backward search. If, at the end, there are state-actions pairs that are not removed from π by the backward search, then this means that π violates the "fairness assumption" for strong-cyclic planning; i.e., there is a cycle induced by π from which there is no possibility of reaching to the goal states or

to the $OPEN$ states, which, in effect, is the same as the former. In this case, NoGood returns TRUE, forcing FS^3 to backtrack. Otherwise, it returns FALSE.

The NoGood function for the strong-cyclic planning is defined as follows:

```
Procedure NoGood(π, S_OPEN, G, S_0) /* for Strong-Cyclic Planning */
  S' ← ∅; S ← G ∪ S_OPEN
  while S' ≠ S
    S' ← S
    S ← S ∪ {s' | (s', a) ∈ π, and S ∩ γ(s', a) ≠ ∅}
    π ← π \ {(s, a) | s ∈ S and (s, a) ∈ π}
  if S_0 ⊆ S and π = ∅ then return FALSE
  return TRUE
```

4.4 Symbolic Model-Checking Primitives in FS^3

This section presents a framework for implementing the data structures of the FS^3 procedure and its helper routines using BDD-based symbolic model-checking primitives. This framework uses the same machinery to represent the states of a planning domain as in [CPRT03]. This machinery is based on using propositional formulae to compactly represent sets of states and possible transitions between those states in a planning domain.

I assume a vector $\overline{s}$ of propositions that represents the current state of the world. For example, in the Hunter-Prey world with a 3×3 grid and one prey, $\overline{s}$ is $\{hx = 0, \ldots, hx = 3, hy = 0, \ldots, hy = 3, px = 0, \ldots, px = 3, py = 0, \ldots, py = 3, prey\_caught\}$. A state is an assignment of the truth-values $\{$TRUE,FALSE$\}$ to each proposition in $\overline{s}$. Let $s(\overline{s})$ denote such an assignment.

Based on this formulation, a set of states S corresponds to the formula $S(\overline{s})$ such that

$$S(\overline{s}) = \bigvee_{s \in S} s(\overline{s}).$$

This definition of set of states is the basis of our framework in this paper. It allows us to define FS^3's forward search mechanism over BDD-based representations of sets of states, rather than single states.

I also assume another vector $\overline{s}'$ of propositional variables to represent the next states of the world, respectively. Similarly, a vector $\overline{a}$ of action variables represents a set of actions to be applied at the same time. A policy π, which is a set of state-action pairs, can be represented as a formula $\pi(\overline{s}, \overline{a})$ in the variables $\overline{s}$ and $\overline{a}$. The formula $S(\overline{s})$ denotes a set of states S in the state vector $\overline{s}$ as before. A situation is represented as a pair of the form $(S(\overline{s}), \chi)$, where χ is the search-control information, as described previously.

The initial situation can be represented by $\{(S_0(\overline{s}), \chi)\}$, where $S_0(\overline{s})$ represents the initial set of states and χ is the initial search-control information. Similarly, the formula $G(\overline{s})$ represents the set of goal states. I assume the existence of a state-transition relation R, which can be represented as $R(\overline{s}, \overline{a}, \overline{s}')$, where $\overline{s}$ denotes the current state vector, $\overline{a}$ denotes the current action vector, and $\overline{s}'$ denotes the next state vector. Note that R is an equivalent representation of the state-transition function γ in a planning domain.

The formulations of the inequality of sets, set difference operations, and subset relations constitute the most basic primitives used in conditionals and termination conditions of the loops of our algorithms. These operations can be easily encoded in terms of basic logical operations on the formulas described above. The inequality of two sets of states can be represented as the formula

$$\neg(S(\overline{s}) \Leftrightarrow S'(\overline{s})),$$

where $S(\overline{s})$ and $S'(\overline{s})$ are the two formulas representing the two sets of states under con-

sideration. Similarly, the subset relation between two sets of states corresponds to the following formula:

$$S(\overline{s}) \implies S'(\overline{s}).$$

A set difference operation $S \setminus S'$ over two sets of states, S and S' can be represented as follows:

$$S(\overline{s}) \wedge \neg S'(\overline{s}).$$

The result of applying an action a in a set of states S can be represented as the formula:

$$\exists \overline{s}' : S(\overline{s}) \wedge R(\overline{s}, \overline{a}, \overline{s}')[\overline{s}'/\overline{s}],$$

where $[\overline{s}'/\overline{s}]$ is called the *forward-shifting* operation [CPRT03]. Note that the above formula represents the $succ(S, a)$ function described in the previous section.

The StatesOf primitive used for computing the set of all states described by a set of situations can be represented as a set-union operator over the situations we are interested in. More formally, the computation of the set of all states of $OPEN = \{x_1, x_2, \ldots, x_n\}$ corresponds to the formula $S_1(\overline{s}) \vee S_2(\overline{s}) \vee \ldots \vee S_n(\overline{s})$, where $x_i = (S_i, \chi_i)$.

The check for cyclic and goal states in the states of a situation is built on set-difference and set-union operations, which can be represented as follows: $S(\overline{s}) \wedge \neg(G(\overline{s}) \vee \exists \overline{a} : \pi(\overline{s}, \overline{a}))$.

The NoGood functions for strong and strong-cyclic planning are based on two primitives for computing WeakPreimage and StrongPreimage of a particular set of states. These preimage computations correspond to

$$\exists \overline{a} \exists \overline{s}'.\pi(\overline{s}, \overline{a}) \wedge S(\overline{s}') \text{ and } \exists \overline{a} \exists \overline{s}'.\pi(\overline{s}, \overline{a}) \implies S(\overline{s}'),$$

respectively.

The composition of two situations that have the same search-control information is a set-union operation over the sets of states described by those situations. In other words, if two situations $x_1 = (S_1, \chi)$ and $x_2 = (S_2, \chi)$ are to be composed into a situation x, then the situation x is the result of the following computation: $x = (S_1(\overline{s}) \vee S_2(\overline{s}), \chi)$. The Compose procedure of FS^3 traverses a given set of situations and composes the appropriate ones by using the computation above.

Finally, the update of a policy π by a set of state-action pairs π' is represented as follows: $\pi(\overline{s}, \overline{a}) \vee \pi'(\overline{s}, \overline{a})$.

4.5 Formal Properties

This section presents theorems showing the completeness and the correctness of the FS^3 planning procedure for nondeterministic planning domains. The proofs are given in the Appendix A.2.

The FS^3 planning procedure starts from the initial states of the input planning problem and performs a forward search towards the goals. FS^3 always terminates: the size of $OPEN$ set cannot grow unboundedly, as there are only a finite number of possible state transitions and $OPEN$ becomes the empty set after finitely many iterations, as FS^3 removes the situations that contain only the goal and the visited states from $OPEN$ at each iteration.

Theorem 9 *The planning procedure FS^3 always terminates.*

The correctness of FS^3 depends on the correctness of the NoGood function, as this

function eliminates the partial policies that cannot be extended to a solution for the input weak, strong, or strong-cyclic planning problem. The backward search of the NoGood functions defined in the previous section always eliminates a partial policy that cannot be a solution, as the NoGood functions are basically simple modifications of the model-checking based weak, strong, and strong-cyclic planning algorithms of [CPRT03], which have been shown to be correct.

Theorem 10 *Let $P = (\Sigma, S_0, G)$ be a planning problem in a nondeterministic planning domain Σ, and let π be a partial policy in Σ If π is a candidate solution for P, then an invocation of NoGood(π, S_π^t, G, S_0) returns* FALSE*, where S_π^t are the terminal states of π. Otherwise, NoGood(π, S_π^t, G, S_0) returns* TRUE*.*

The following theorem establishes the correctness of the FS^3 planning procedure:

Theorem 11 *Suppose one of the search traces of FS^3 returns a policy π given the input planning problem $P = (\Sigma, S_0, G)$ in a nondeterministic planning domain Σ. Then π is a solution for the planning problem P.*

Finally, the following theorem establishes the completeness of FS^3:

Theorem 12 *Suppose $P = (\Sigma, S, G)$ is a χ-solvable nondeterministic planning problem given the search-control information χ. Then, one of the search traces of FS^3 returns a solution policy for P using χ.*

4.6 Examples

This section describes two planning algorithms that are instances of the abstract FS^3 procedure. The first algorithm, FS$^3_{\text{TLPlan}}$, combines temporal-logic based search-control rules as in ND-TLPlan with BDD-based symbolic model-checking primitives to compactly represent sets of states during planning. The second algorithm, FS$^3_{\text{SHOP2}}$, does a similar combination of ND-SHOP2's HTNs and BDD-based representations.

4.6.1 FS^3 with Control Rules

The FS$^3_{\text{TLPlan}}$ planning algorithm uses temporal-logic based control rules to specify search-control information as in ND-TLPlan described in the previous chapter. Figure 4.5 shows the pseudocode of this algorithm. FS$^3_{\text{TLPlan}}$ successively progresses the input temporal-logic (TL) formula over BDD-based state representations as follows. The input to the planner is the initial $OPEN$ set, the set of goal states, and the empty policy. At each iteration, the planning algorithm first removes the cyclic and goal states from the $OPEN$ situations as described for the abstract FS^3 procedure. Then, it performs the NoGood correctness test on the $OPEN$ situations and the current partial policy as described in the previous sections.

In Figure 4.5, the Control function is responsible for implementing both the search-control function acceptable and the progression function progress for FS$^3_{\text{TLPlan}}$'s search-control information encoded as TL formulas. The formal definition of the Control function in FS$^3_{\text{TLPlan}}$ is given in Figure 4.6. It splits the set S of states by generating an action a that is applicable in some of the states in S and that is acceptable with

```
Procedure FS^3_TLPlan(OPEN, G, π)
  OPEN ← {(S \ (G ∪ S_π), χ) | (S, χ) ∈
OPEN and S \ (G ∪ S_π) ≠ ∅}
  if NoGood(π, StatesOf(OPEN), G, S_0) then return(FAILURE)
  if OPEN = ∅ then return(π)
  select a situation (S, χ) from OPEN and remove it
  F ← Control(S, χ)
  if F = ∅ then return(FAILURE)
  OPEN' ← Compute-Successors(F, OPEN)
  π' ← π ∪ {(s, a) | (S', a, χ') ∈ F and s ∈ S'}
  return( FS^3_TLPlan(OPEN', G, π'))
```

Figure 4.5: $\mathsf{FS}^3_{\mathsf{TLPlan}}$, an instance of the abstract FS^3 procedure that uses search-control rules as in ND-TLPlan. In the initial call of the algorithm, π is the empty policy and $OPEN$ is the set that contains only the initial situation (S_0, χ_0).

```
Procedure Control(S, χ)
  F ← ∅;  loop
    if S = ∅ then return(F)
    nondeterministically choose an action a such that S ∩ S_a ≠ ∅
    χ' ← PROGRESS(S ∩ S_a, χ)
    if χ' = FALSE then return ∅
    F ← F ∪ {(S ∩ S_a, a, χ')}
    S ← S \ S_a
  return F
```

Figure 4.6: The Control procedure.

respect to the current search-control information χ. Then, $\mathsf{FS}^3_{\mathsf{TLPlan}}$ generates the next search-control information χ' by using the PROGRESS function shown in Figure 4.6. PROGRESS is the original progression function of TLPlan and ND-TLPlan, except that it attempts to satisfy a logical condition in a set of states represented as a BDD, rather than over a single state. The logical condition that needs to be satisfied is, in this case, the non-temporal subformula in χ, and the set of states χ needs to be satisfied in is $S \cap S_a$ – i.e., the states where the current action a is applicable. In order to perform such a satisfia-

bility check, the non-temporal subformula in χ is converted into a BDD that represents all of the states, say S_χ, in which that subformula is satisfied, and the satisfiability check is performed simply as checking the subset relation: $(S \cap S_a) \subseteq S_\chi$. If $(S \cap S_a) \subseteq S_\chi$ then this means that the subformula of χ is satisfied in every state where the current action is applicable, and therefore, PROGRESS returns TRUE. Otherwise, it returns FALSE.

During its search, if PROGRESS generates a temporal-logic formula that is the logical constant FALSE then Control returns the empty set, forcing $\mathsf{FS}^3_{\mathsf{TLPlan}}$ to fail in the current search trace. Otherwise, Control continues with those states of S in which the action a is not applicable in order to generate another action for those states. This search continue until Control plans an action for every state in the input set of states S or it returns FAILURE at some iteration as described above.

The symbolic representations of the set-based operations in Control is the same as the ones described for FS^3 in the previous section.

4.6.2 FS^3 with Hierarchical Task Networks

$\mathsf{FS}^3_{\mathsf{SHOP2}}$ combines task-decomposition methods as in ND-SHOP2's HTNs with BDD-based state representations. Figure 4.7 shows the pseudocode of the planning algorithm. $\mathsf{FS}^3_{\mathsf{SHOP2}}$ does successive task decompositions over BDD-based representations of classes of states as follows. The input to the planner is the initial $OPEN$ set, the set of goal states, and the empty policy. At each iteration, the planner first removes the cyclic and goal states from the $OPEN$ situations as described for the FS^3 procedure. Then, it performs the aforementioned NoGood correctness test on the $OPEN$ situations and the

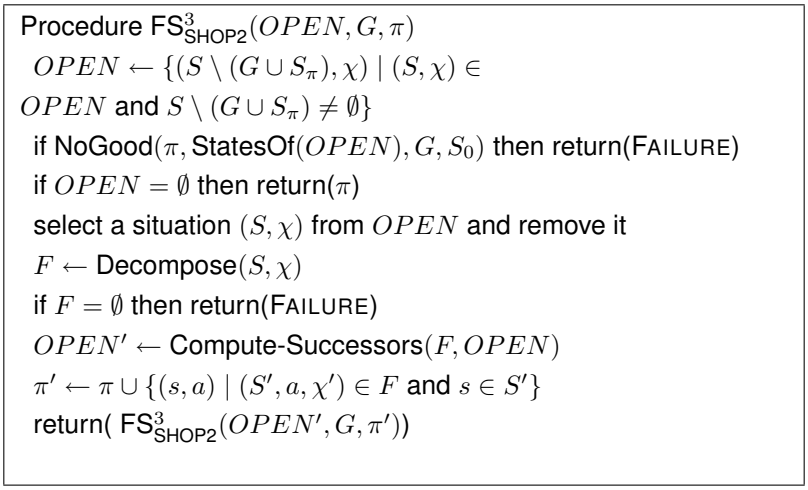

Figure 4.7: FS$^3_{\text{SHOP2}}$, an instance of the abstract FS^3 procedure that uses HTNs as in ND-SHOP2. In the initial call of the algorithm, π is the empty policy and $OPEN$ is the set that contains only the initial situation (S_0, χ_0).

current partial policy.

In Figure 4.7, the Decompose function is responsible for implementing the search-control function acceptable and the progression function progress for FS$^3_{\text{SHOP2}}$'s search-control information encoded as HTNs. Intuitively, in a set S of states represented as a BDD, a possible HTN decomposition of a task t specifies (1) a set of subtasks and (2) a subset S' of S in which the particular decomposition of t is possible. Thus, decomposing a task t in a set S of states represented by a BDD yields two sub-BDDs — one that represents S' and and the other that represents the rest of the states $S \setminus S'$ in which other possible decompositions for t must be tried.

The formal definition of the Decompose function is given in Figure 4.8. In a situation (S, χ), let t be a task that has no predecessors in the task network χ. If t is a primitive task then t can be executed directly in the world. Let a be an action that corresponds to t, and a can be applied in each state in S; i.e., $S \subseteq S_a$. Note that applying an action a in a set of states S does not generate any new open situations: that is, S must

```
Procedure Decompose(S, χ)
 F ← ∅; X ← {(S, χ)}
 loop
  if X = ∅ then return(F)
  select a tuple (S, χ) ∈ X and remove it
  select a task t that has no predecessors in w
  if t is a primitive task then
     actions ← {a | a ∈ A is an action for t, and S ⊆ S_a}
     if actions = ∅ then return ∅
     select an action a from actions
     F ← F ∪ {(S, a, χ \ {t})}
  else
     methods ← {m | m is a task-decomposition method for t
                    and S ∩ S_m ≠ ∅}
     if methods = ∅ then return ∅
     select a method instance m from methods
     X ← X ∪ {(S ∩ S_m, (χ \ {t}) ∪ χ')}
     if S \ S_m ≠ ∅ then X ← X ∪ {(S \ S_m, χ)}
```

Figure 4.8: The Decompose procedure.

be a subset of S_a because, otherwise, there is at least one state in S for which no action is applicable, and this is a failure point in planning.

If t is not primitive, then Decompose successively applies methods to the non-primitive tasks in χ until an action is generated. Suppose it chooses to apply a method m to t. Let S_m be the set of all states in which m is applicable to t. This generates two possible situations: (1) the situation that arises from decomposing t by m in the states $S \cap S_m$ in which m is applicable, and (2) the situation that specifies the states in which m is not applicable – i.e., the situation $(S \setminus S_m, \chi)$. In the former case, Decompose proceeds with decomposing the subtasks of t as specified in m. In the latter case, on the other hand, other methods for t must be used. Note that if there are no other methods for t to be used in situations like $(S \setminus S_m, \chi)$, then Decompose returns the empty set.

Decompose returns a set F of the form $\{(S_i, a_i, \chi_i)\}_{i=0}^{k}$. If $F = \emptyset$ then this means that the decomposition process has failed since there is a state $s \in S$ such that there is no action for s that can be generated by using the methods provided for the underlying planning domain. If $F \neq \emptyset$ then the routine has generated an action a_i for each state in S — i.e., $S = \bigcup_i S_i$ —, and a task network χ_i to be accomplished after applying that action.

The symbolic representations of the set-based operations in Decompose is the same as the ones described for FS^3, except that the check whether a method or an action is applicable in a given set S of states corresponds to the following formula: $S(\overline{s}) \implies S_a(\overline{s})$ and $S(\overline{s}) \wedge S_m(\overline{s})$, where $S(\overline{s})$ represents the set of states in which Decompose is performing these checks, and $S_a(\overline{s})$ and $S_m(\overline{s})$ represents the set of all states in which the action a and the method m is applicable.

4.7 Experimental Evaluation

This section describes an extensive experimental comparison of the FS$^3_{\text{SHOP2}}$ planning algorithm, one of the instances of FS^3 as described above, with the ND-SHOP2 and MBP planning systems. The current implementation of FS$^3_{\text{SHOP2}}$ is built on both the ND-SHOP2 and the MBP planning systems. It differs from ND-SHOP2 in three ways: (1) it plans over sets of states rather than a single state, and (2) it includes the NoGood routine as a part of its backtracking search, and (3) it implements an interface to MBP for exploiting the machinery of BDDs implemented in it.

The experimental evaluation of FS$^3_{\text{SHOP2}}$ consists of three sets of experiments in the Hunter-Prey domain. In these experiments, the domain was fully-observable in the

sense that the hunter can always observe the location of the prey. The hunter moves first in the world, and the prey moves afterwards. In the domain representation, the prey does not appear as a separate agent. Instead, the prey's possible actions are encoded as the nondeterministic outcomes for the hunter's actions. As before, all three planners, $\mathsf{FS}^3_{\mathsf{SHOP2}}$, ND-SHOP2 and MBP, were provided as input the same planning operator descriptions (i.e., action descriptions) for the Hunter-Prey planning domain in their respective representational languages. Both $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and ND-SHOP2 were provided the same search-control information encoded as hierarchical task networks for this domain.

All experiments were run on an AMD Duron 900MHz laptop with 256MB memory, running Linux Fedora Core 2. If a planning algorithm failed on a problem (i.e., it ran out of memory or it could not solve the problem within a time limit of 40 minutes), it was run again on another problem of the same size. Each data point on which the planning algorithm failed more than five times is omitted from the experimental results, but those data points where it failed 1 to 5 times are included. Thus the experimental results make the performance of ND-SHOP2 and MBP look better than it really was—but this makes little difference since they performed much worse than $\mathsf{FS}^3_{\mathsf{SHOP2}}$.

Experimental Set 1. These experiments aimed to investigate how well $\mathsf{FS}^3_{\mathsf{SHOP2}}$ is able to cope with large-sized problems compared to ND-SHOP2 and MBP. To achieve this objective, the experiments are done with hunter-prey problems with increasing grid sizes and with only one prey so that the nondeterminism in the world is kept at a minimum for the hunter.

Figure 4.9 shows the results of the experiments for grid sizes $n = 5, 6, \ldots, 10$. For

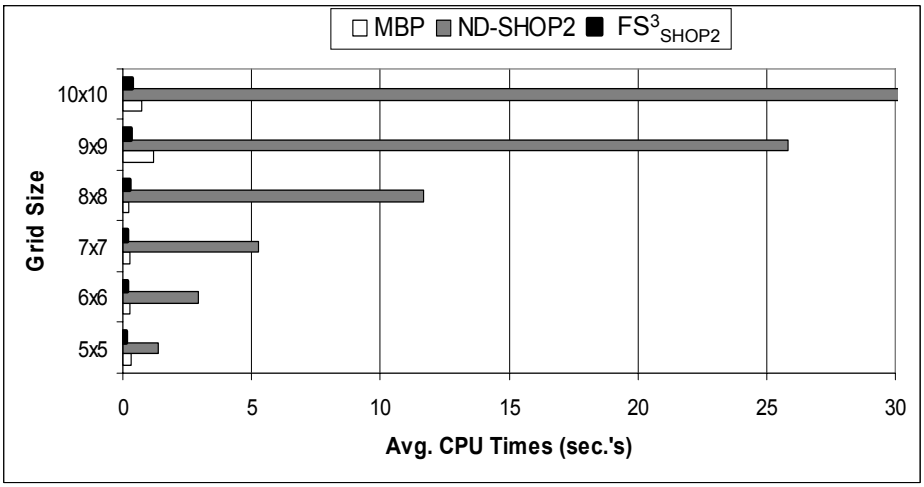

Figure 4.9: Average running times (in sec.'s) of $\mathsf{FS}^3_{\mathsf{SHOP2}}$, ND-SHOP2, and MBP in the Hunter-Prey domain as a function of the grid size, with one prey.

each value for n, MBP, ND-SHOP2, and $\mathsf{FS}^3_{\mathsf{SHOP2}}$ were run on 20 randomly-generated problems. This figure reports the average running times required by the planners on those problems. For grids larger than $n = 10$, ND-SHOP2 was not able to solve the planning problems due to memory overflows. This is because the sizes of the solutions in this domain are very large, and therefore, ND-SHOP2 runs out of memory as it tries to store them explicitly. Note that this domain admits only high-level search strategies such as "look at the prey and move towards it." Although this strategy helps the planner prune a portion of the search space, such pruning alone does not compensate for the explosion in the size of the explicit representations of the solutions for the problems.

On the other hand, both $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and MBP was able to solve all of the problems in these experiments. The difference between the performances of $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and ND-SHOP2 demonstrates the impact of the use of BDD-based representations: $\mathsf{FS}^3_{\mathsf{SHOP2}}$, using the same HTN-based heuristic as ND-SHOP2, was able to scale up as good as MBP

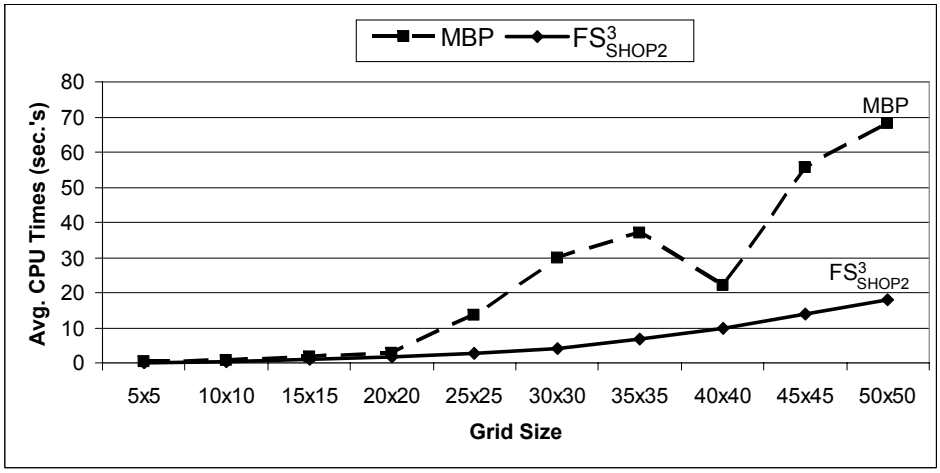

Figure 4.10: Average running times (in sec.'s) for $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and MBP on larger problems in the Hunter-Prey domain as a function of the grid size, with one prey.

since it is able to exploit BDD-based representations of the problems and their solutions.

In order to see how $\mathsf{FS}^3_{\mathsf{SHOP2}}$ performs in larger problems compared to MBP, I have also experimented with $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and MBP in much larger grids. Figure 4.10 shows the results of these experiments with varying the size of the grids in the planning problems as $n = 5, 10, 15, \ldots, 45, 50$.

These results show that $\mathsf{FS}^3_{\mathsf{SHOP2}}$ is able to perform better than MBP with the increasing grid size. The running times required by both of the planners increase in larger grids; however, this increase is much slower for $\mathsf{FS}^3_{\mathsf{SHOP2}}$ than MBP as shown in Figure 4.10 due to the following reasons: (1) $\mathsf{FS}^3_{\mathsf{SHOP2}}$ is able to combine the advantages of exploiting HTN-based search-control heuristics with the advantages of using BDD-based representations, whereas MBP cannot exploit HTN-based strategies to complement its BDD-based planning techniques; and (2) $\mathsf{FS}^3_{\mathsf{SHOP2}}$, being a forward planner, considers only those states that are reachable from the initial states of the planning problems, whereas MBP's backward-chaining algorithms explore states that are not reachable from

the initial states of the problems at all.

In Figure 4.10, the data-point for MBP in the experiments with 40×40 grids shows an unexpected decline in the performance of the planning system. With the same experimental parameters and setup, the neighboring data points for grids 35×35 and 45×45, respectively, does not reflect this anomaly. I ran the experiments with 40×40 grids with different random problem sets three times and the results were always the same. My speculation is that the anomaly might be occurring due to some problem in MBP's implementation and/or its integration with the CUDD package for BDDs.

Experimental Set 2. In order to investigate the effect of combining search-control strategies and BDD-based representations in $\mathsf{FS}^3_{\mathsf{SHOP2}}$, I used a variation of the Hunter-Prey domain, where there are more than one prey in the world, and the prey i cannot move to any location within the neighborhood of prey $i + 1$ in the world. In such a setting, the amount of nondeterminism for the hunter after each of its move increases combinatorially with the number of preys in the domain. Furthermore, the BDD-based representations of the underlying planning domain explode in size under these assumptions, mainly because the movements of the preys are dependent to each other.

In this adapted domain, ND-SHOP2 and $\mathsf{FS}^3_{\mathsf{SHOP2}}$ were provided with a search-control strategy that tells the planners to chase the first prey until it is caught, then the second prey, and so on, until all of the preys are caught. Note that this heuristic allows for abstracting away from the huge state space: when the hunter is chasing a prey, it does not need to know the locations of the other preys in the world, and therefore, it does not need to reason and store information about those locations.

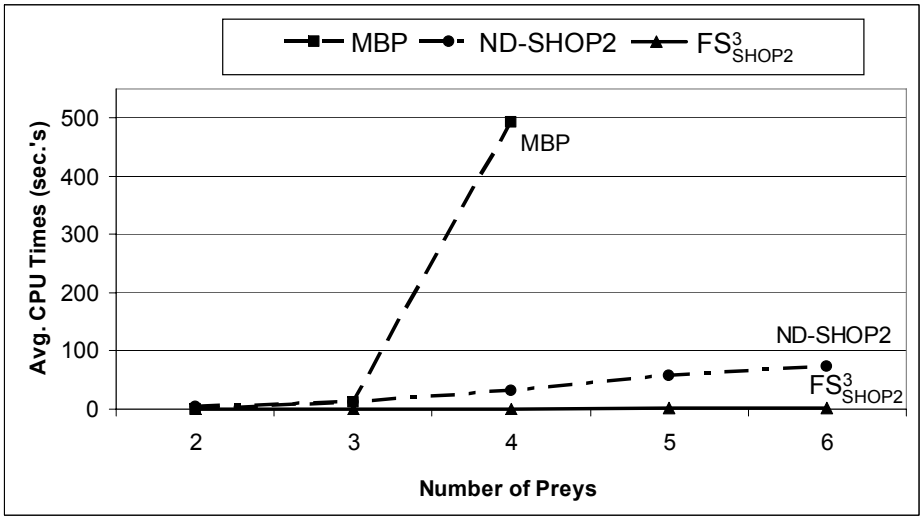

Figure 4.11: Average running times (in sec.'s) of ND-SHOP2, $\text{FS}^3_{\text{SHOP2}}$ and MBP on problems in the Hunter-Prey domain as a function of the number of preys, with a 4×4 grid. MBP was not able to solve planning problems with 5 and 6 preys within 40 minutes.

The experiments in this set aimed to investigate the running times of MBP, ND-SHOP2, and $\text{FS}^3_{\text{SHOP2}}$, with varying the number of preys from $p = 2, \ldots, 6$ in a 4×4 grid world. Figure 4.11 shows the results. Each data point is an average of the running times of all three planners on 20 randomly-generated problems for each experiment with different numbers of prey. These results demonstrate the power of combining HTN-based search-control heuristics with BDD-based representations of states and solutions in our planning problems: $\text{FS}^3_{\text{SHOP2}}$ was able to outperform both ND-SHOP2 and MBP. The running times required by MBP grow exponentially faster than those required by $\text{FS}^3_{\text{SHOP2}}$ with the increasing size of the preys, since MBP cannot exploit HTN-based heuristics. Note that ND-SHOP2 performs much better than MBP in the presence of good search-control heuristics.

Experimental Set 3. The final set of experiments are designed to further investigate $\mathsf{FS}^3_{\mathsf{SHOP2}}$'s performance compared to that of ND-SHOP2 and MBP on Hunter-Prey problems, with multiple preys and with increasing grid sizes. In these experiments, the number of preys were varied as $p = 2, \ldots, 6$ and the grid sizes were varied as $n = 3, 4, 5, 6$.

Table 4.3 reports the average running times required by $\mathsf{FS}^3_{\mathsf{SHOP2}}$, MBP, and ND-SHOP2 in these experiments. Each data point is an average of the running times of all three planners on 20 randomly-generated problems for each experiment with different p and n combinations. These results provide further proof for our conclusions. Search-control heuristics helped both $\mathsf{FS}^3_{\mathsf{SHOP2}}$ and ND-SHOP2 as they both outperform MBP with the increasing number of the preys. However, with increasing grid sizes, ND-SHOP2 runs into memory problems as before due to its explicit representations of states and solutions of the problems. $\mathsf{FS}^3_{\mathsf{SHOP2}}$, on the other hand, was able to cope with very well both with increasing the grid sizes and the number of preys in these problems.

These experimental results demonstrate the importance of using search-control heuristics and BDD-based representations in a single forward-chaining framework. The search-control heuristics exploited the structure of the underlying planning problems, and therefore, they resulted in a more compact and structured BDD representations of the planning problems and domains. For example, in the hunter-prey domain, the strategy, which tells $\mathsf{FS}^3_{\mathsf{SHOP2}}$ to focus on catching one prey while ignoring other preys, provides a combinatorial reduction in the representations of the solutions for the problems and the state-transition relation for the domain. BDDs provide even further compactness in those reduced representations. Note that the same strategy did not work for ND-SHOP2 very

Table 4.3: Average running times (in sec.'s) of MBP, ND-SHOP2, and FS^3_{SHOP2} on Hunter-Prey problems with increasing number of preys and increasing grid size.

	2 preys		
Grid	**MBP**	**ND-SHOP2**	**FS^3_{SHOP2}**
3x3	0.343	0.78	0.142
4x4	0.388	3.847	0.278
5x5	1.387	18.682	0.441
6x6	3.172	76.306	0.551
	3 preys		
Grid	**MBP**	**ND-SHOP2**	**FS^3_{SHOP2}**
3x3	1.1	1.72	0.329
4x4	11.534	12.302	0.521
5x5	133.185	58.75	0.92
6x6	368.166	250.315	1.404
	4 preys		
Grid	**MBP**	**ND-SHOP2**	**FS^3_{SHOP2}**
3x3	29.554	3.256	0.448
4x4	492.334	31.591	0.759
5x5	>40 mins	176.49	1.818
6x6	>40 mins	547.911	3.295
	5 preys		
Grid	**MBP**	**ND-SHOP2**	**FS^3_{SHOP2}**
3x3	233.028	5.483	0.655
4x4	>40 mins	56.714	1.275
5x5	>40 mins	304.03	3.028
6x6	>40 mins	memory-overflow	7.059
	6 preys		
Grid	**MBP**	**ND-SHOP2**	**FS^3_{SHOP2}**
3x3	2158.339	8.346	0.781
4x4	>40 mins	73.435	1.786
5x5	>40 mins	486.112	5.221
6x6	>40 mins	memory-overflow	11.826

well in large problems due to explicit representations of the problems and the domain. Note also that BDD-based representations alone did not work very well for MBP in problems with increasing number of the preys, since those representations are not sufficient to abstract away from the irrelevant portions of the state space.

Chapter 5

Forward-Chaining Planning with MDPs

Planning algorithms for MDPs typically have large efficiency problems due to the need to explore all or most of the state space. For complex planning problems, the state space can be quite huge. This chapter continues the discussion on using search control in planning under uncertainty and describes a way to improve the efficiency of planning on MDPs by adapting the search-control (i.e., pruning) techniques used in forward-chaining classical planners described previously.

In particular, Sections 5.1 and 5.2 describe how to modify any forward-chaining MDP planning algorithm, by incorporating into it the search-control function from any forward-chaining classical planner. Section 5.3 describes conditions under which the modified MDP planning algorithms are guaranteed to find optimal answers, and conditions under which they can do so exponentially faster than the original MDP planners. Section 5.4 presents an experimental evaluation of the modified versions of Real-Time Dynamic Programming (RTDP) [BG00], Labeled RTDP (LRTDP) [BG03], and a forward-chaining Value Iteration (Value Iteration) [Ber05] algorithm. In these experiments, modified algorithms ran exponentially faster than the original ones. On the largest problems the original algorithms could solve, the modified ones ran about 10,000 times faster. In only about 1/3 second, the modified algorithms could solve problems whose state spaces were more than 14,000 times larger.

```
Procedure Forward-VI
 select any initialization for the value function V
 while V has not converged do
  S ← S_0; Visited ← ∅; π ← ∅
  while S ≠ ∅ do
    for every state s ∈ S ∩ G, V(s) ← R(s)
    S ← S \ G; S' ← ∅
    for every state s ∈ S
      for every a ∈ app(s)
        Q(s,a) ← C(s,a) + α Σ_{s'∈results(s,a)} Pr(s,a,s') V(s')
        S' ← S' ∪ {s | s ∈ results(s,a)}
      V(s) ← min_{a∈app(s)} Q(s,a)
      a ← argmin_{a∈app(s)} Q(s,a)
      π ← π ∪ {(s,a)}
    Visited ← Visited ∪ S
    S ← S' \ Visited
 return π
```

Figure 5.1: Forward-VI, a forward-chaining version of Value Iteration.

5.1 Forward-Chaining MDP Planners

Many existing MDP planning algorithms can be viewed as forward-search procedures embedded inside iteration loops. Examples include traditional Value Iteration algorithm [Ber05], RTDP [BG00], LRTDP [BG03], and *heuristic search techniques* such as LAO* [HZ01]. The forward search in these MDP planners starts at the initial states and searches forward by applying actions to states, computing a policy and/or a set of utility values as the search progresses. The iteration loop continues until some sort of convergence criterion is satisfied (e.g., until two successive iterations produce identical utility values for every node, or until the residual of every node becomes less than or equal to a termination criterion $\epsilon > 0$).

As an example, the well-known Value Iteration algorithm [Ber05] can be seen as

a forward-search procedure by making sure that in each iteration, Value Iteration starts computing its values from the initial states of an MDP planning problem and proceeds toward the goals. Figure 5.1 shows the pseudocode of a forward-chaining version of Value Iteration. The iteration loop is the outer while loop, and the forward-search procedure is everything inside that loop. The planning process continues until the value function V converges to its optimal form as in the traditional description of the Value Iteration algorithm. In the pseudocode, app is the set of applicable actions in a state s: i.e.,

$$\mathsf{app}(s) = \{a \mid a \in A \text{ and } \gamma(s, a) \neq \emptyset\}.$$

Planners like RTDP [BG00] and LRTDP [BG03] fit directly into the above format. For example, Figure 2.5 in Section 2.3.1 shows the pseudocode for RTDP. RTDP is a planning algorithm based on *real-time* forward search [Kor90] that is performed by simulating possible executions of the actions in the states visited during search, rather than by exploring all or most of the states that can be reached from the initial states of the input MDP planning problem.

In each iteration of the outer while loop, RTDP performs a greedy search going forward starting from the initial state towards the goal states of the input MDP planning problem. As described in Section 2.3.1, RTDP's forward search at each iteration is a stochastic simulation of the greedy partial policy: in a state s, RTDP chooses an action a that has currently the best $Q(s, a)$ value. Then the algorithm does a one-step update of this $Q(s, a)$ by using the Bellman Equation (Eqs. 2.1 and 2.2). Next, it generates a successor state of s by probabilistically sampling the state transitions induced by applying a in s using the transition probabilities as specified by Pr.

LRTDP is a variant of RTDP that implements a labeling mechanism in order to mark the states whose values are converged during planning so that those states are not expanded by the planning algorithm again. Note that in order the value of a state to be converged, the values of all of its descendants need to be converged during planning. LRTDP is shown to be correct — i.e., the labeling mechanism does not eliminate any optimal solutions —, and in some cases, it outperforms RTDP.[1]

LAO* is a heuristic search algorithm based on a generalization of the well-known AO* search [Nil80]. It extends AO* to incorporate mechanisms to deal with cyclic search traces. Similar to the above planners, LAO* consists of an iteration loop in which the algorithm performs a forward search starting from the initial states toward the goal states of the input MDP planning problem. Unlike the previous planners, LAO* interleaves its forward search with a dynamic programming step as follows. In each iteration, LAO* first generates the best partial policy in a forward fashion. Then, the planning algorithm uses dynamic programming to update the value of each state that is in that policy or that is an ancestor of a state in that policy in the state space. The iteration continues until the difference between the values of the states in the successive best policies falls below an error threshold.

[1]Note that LRTDP is not guaranteed to perform better than RTDP in general. The reason for this is that, in order to preserve correctness, a state must be labeled as solved only if all of its π-descendants are labeled as solved in a greedy policy π. There are planning problems where the value function for some states converge towards the end of planning process, and in such cases, LRTDP performance is not higher than that of RTDP.

5.2 Modifying MDP Planners with Search Control

At each state s that an MDP planner visits during its forward search, the planner needs to know app(s), the set of all actions applicable to s. For example, the innermost for loop of Forward-VI iterates over the actions in app(s); RTDP and LRTDP choose whichever action in app(s) currently has the best value and simulate its effects; and LAO* expands one of the terminal states in the current best policy by choosing an action in app(s).

The rest of this section describes how to modify the forward search in MDP planners in order to incorporate the search control techniques originally developed for classical planning algorithms. The modifications are based on the acceptable and progress functions of the abstract planning procedure FCP presented Section 3.2 (see Figure 3.2). Recall that FCP is a simple forward state-space search procedure that can use auxiliary search-control information χ in order to guide its search. Its search-control function acceptable is responsible for pruning some of the actions in app(s) in a state and its progress function generates the search-control information to be used in a successor state that arise from applying an acceptable action in the current state.

Let $\Sigma = (S, A, \gamma)$ be a classical planning domain and let $\Sigma' = (S', A', \gamma', \alpha, Pr, C)$ be an MDP. Let P be a classical planning problem $P = (\Sigma, s_0, G)$ in Σ and let $P^F = (\Sigma', S_0, G)$ be an MDP planning problem in Σ'. The MDP planning problem P^F is an *MDP version* of the classical planning problem P if and only if the following holds:

- $S = S'$ and $s_0 \in S_0$;
- there is a one-to-one mapping det from A' to A such that the following holds:

– for each state $s \in S$, if $\gamma'(s, a') = \emptyset$ then $\gamma(s, det(a')) = \emptyset$.

– Otherwise, $\gamma(s, det(a')) \in \gamma'(s, a')$.

Intuitively, P^F is an ***MDP version*** of P if and only if the two planning problems have the same states and goals, the initial state of P is among the possible initial states of P^F, and if there is a one-to-one mapping det from P^F's actions to P's actions such that for every action a in P^F, a and $det(a)$ are applicable to exactly the same states, and for each such state s, $\gamma(s, det(a)) \in \gamma'(s, a)$. The additional states in $\gamma'(s, a)$ are used to model various sources of the uncertainty in the domain, such as action failures (e.g., a robot gripper may drop its load) and exogenous events (e.g., a road is closed). The action $det(a)$ is called the *deterministic version* of a, and a the ***MDP version*** of $det(a)$.

Let Z be a forward-chaining MDP planning algorithm and let F be a classical planning algorithm that is an instance of FCP. Then, Z^F is a modified version of Z in which every occurrence of $\mathsf{app}(s)$ is replaced by

$$\{a \in \mathsf{app}(s) \mid \mathsf{acceptable}_F(s, det(a), \chi) \text{ holds}\},$$

where χ is the auxiliary search-control information and $det(a)$ is the deterministic version of an action a as described above. The search-control information χ is computed by progression using F's progress function. Each time the forward search in the MDP algorithm Z applies an action a in a state s and generates the successor state s', $\mathsf{progress}(s, a, \chi)$ is the search-control information χ' to be used with the state s'. This is precisely the reason that we require Z to be a forward-chaining algorithm: the auxiliary search-control information χ' is computed by progression from s''s parent.

The following gives some examples of Z^F; in particular, two of the enhanced MDP

planning algorithms, namely Forward-VI$^{\text{TLPlan}}$ and RTDP$^{\text{SHOP2}}$, that are produced by incorporating search control as in TLPlan and SHOP2 (described in Chapter 3) in their original versions.

5.2.1 Forward-VI$^{\text{TLPlan}}$

The first example for Z^F is Forward-VI$^{\text{TLPlan}}$, the enhanced version for the Forward-VI algorithm that incorporates TLPlan's search-control rules. Figure 5.2 shows the pseudocode of this procedure. The algorithm is basically the same as Forward-VI, except that in a state s, it considers only the acceptable actions, rather than all of the applicable ones. The acceptable function is defined using temporal-logic formulas as in TLPlan. In a state s, Forward-VI$^{\text{TLPlan}}$ first checks the search-control formula f that is associated with s. If f is FALSE then this means that the action that is applied to the parent of s yielding s is not an acceptable action. In that case, Forward-VI$^{\text{TLPlan}}$ does not generate any successors of s. Otherwise, it updates the value of s by computing the $Q(s, a)$ values given each applicable action a in s.

The forward search in Forward-VI$^{\text{TLPlan}}$ continues as above until there are no states left to explore in the current iteration of the planning algorithm. Successive iterations of forward search continues until the value function V converges to its optimal value. In practice, the algorithm terminates when the Bellman residual between two successive computations of the value function V is less than or equal to a given ϵ value, which is a very small number. More specifically, Forward-VI$^{\text{TLPlan}}$ terminates when $|V(s) - V'(s)| < \epsilon$ for all states that is reachable by Forward-VI$^{\text{TLPlan}}$'s forward search starting

```
Procedure Forward-VI^TLPlan
 let f be the initial search-control formula
 select any initialization for the value function V
 while V has not converged do
  S ← {(s, f) | s ∈ S_0}; Visited ← ∅; π ← ∅
  while S ≠ ∅ do
    for every (s, f) ∈ S such that s ∈ G, V(s) ← R(s)
    S ← {(s, f) | (s, f) ∈ S and s ∉ G}; S' ← ∅
    for every (s, f) ∈ S
      f' ← Progress(s, f)
      if f' ≠ FALSE then
        for every a ∈ app(s)
          Q(s, a) ← C(s, a) + α Σ_{s'∈results(s,a)} Pr(s, a, s') V(s')
          S' ← S' ∪ {(s', f') | s' ∈ γ(s, a)}
      V(s) ← min_{a∈app(s)} Q(s, a)
      a ← argmin_{a∈app(s)} Q(s, a)
      π ← π ∪ {(s, a)}
    Visited ← Visited ∪ {s | (s, f) ∈ S}
    S ← {s | s ∈ S' and s ∉ Visited}
 return π
```

Figure 5.2: Forward-VI$^{\text{TLPlan}}$, the enhanced version of Forward-VI with TLPlan's control rules.

from the initial states of the input planning problem.

5.2.2 RTDP$^{\text{SHOP2}}$

The second example for Z^F is RTDP$^{\text{SHOP2}}$. Figure 5.3 shows the pseudocode of the enhanced RTDP planning algorithm that incorporates SHOP2's search-control function acceptable and progression function progress, to use HTN-based search-control information as in ND-SHOP2 and FS$^3_{\text{SHOP2}}$.

As RTDP, RTDP$^{\text{SHOP2}}$ performs successive stochastic simulations that start from the initial state towards the goal state. At each state s, however, it successively decom-

Procedure RTDP$^{\text{SHOP2}}$
 select any admissible initialization for V
 while V has not converged relative to a parameter ϵ do
  $s \leftarrow s_0$; $w \leftarrow w_0$
  while $s \notin G$ do
   $W \leftarrow \{w\}$; $A \leftarrow \emptyset$
   while $W \neq \emptyset$ do
    select a task network $w \in W$ and remove it
    $T \leftarrow \{t \mid t \in w$ and t has no predecessors$\}$
    for every task $t \in T$ do
     if t is a primitive task then
      $actions \leftarrow \{(a, \sigma(w - \{t\})) \mid a$ is an action, σ is a substitution s.t.
                $head(a) = \sigma(t),$ and $a \in$ app$(s)\}$
      if $actions = \emptyset$ then return FAILURE
      $A \leftarrow A \cup actions$
     else
      $methods \leftarrow \{(m, \sigma) \mid m$ is an instance of a method in M, σ is a
                substitution s.t. $head(m) = \sigma(t),$ and m is applicable in $s\}$
      for every $(m, \sigma) \in methods$ do
       $w' \leftarrow$ ApplyMethod(s, w, t, m, σ)
       $W \leftarrow W \cup \{w'\}$
   if $A = \emptyset$ then return(FAILURE)
   $(a, w') \leftarrow \text{argmin}_{(a,w') \in A} Q(s, a)$
   $V(s) \leftarrow C(s, a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s, a, s') V(s')$
   pick $s' \in \gamma(s, a)$ with probability $Pr(s, a, s')$
   $s \leftarrow s'$; $w \leftarrow w'$
  extract the greedy optimal policy π given V and s_0
  return π

Figure 5.3: The modified RTDP algorithm that incorporates search control as in SHOP2.

poses the tasks in the current task network w in order to generate all of the actions that are acceptable in s. This computation is similar to the task-decomposition algorithm used in the abstract FS$^3_{\text{SHOP2}}$ planning procedure for generating solutions in nondeterministic planning domains, as described in the previous chapter. Intuitively, this computation first determines a task t that has no predecessors in w. Then, it generates all possible decompositions of t given the task-decomposition methods such that each such decomposition

generates an action a and a successor task network w'. RTDP$^{\text{SHOP2}}$ then chooses the "best" action a among those generated ones and its associated successor task network w'. The best action is the one that has the minimum $Q(s, a)$ value in the current state s, among the actions generated by all possible decompositions of the task t. The algorithm then applies a in s and probabilistically chooses one of the successor states generated by this application. The task network w' is the search-control information to be used along with this successor state in the next iteration of the forward search.

The forward search (i.e., the stochastic simulation) in RTDP$^{\text{SHOP2}}$ continues until a goal state is generated. Then, the algorithm starts a new forward search from the initial state of the input planning problem, unless the value function over all of the states reachable from that initial state is converged. As in Forward-VI$^{\text{TLPlan}}$, an ϵ termination criterion is used in practice.

5.3 Formal Properties of the Modification Technique

Let Z be a forward-chaining MDP planning algorithm that is guaranteed to return an optimal solution if one exists, F be an instance of FCP, and acceptable$_F$ be F's search control function. Suppose $\Sigma = (S, A, \gamma, \alpha, Pr, C)$ is an MDP and $P = (\Sigma, S_0, G)$ be a planning problem in Σ. The succ of a state s in Σ is the set

$$\text{succ}(s) = \{s' \mid a \in applicable(s) \text{ and } s' \in \gamma(s, a)\}.$$

Recall that app(s) of a state s is the set of actions that are applicable in s and $\gamma(s, a)$ is the set of successor states that arise from applying a in s.

Then, I define the *reduced* MDP Σ^F and planning problem P^F as follows:

$$\mathsf{app}^F(s) = \{a \in \mathsf{app}(s) \mid \mathsf{acceptable}_F(s, det(a), \chi) \text{ holds}\};$$

$$Pr^F(s,a,s') = \begin{cases} Pr(s,a,s') & \text{if } a \in \mathsf{app}^F(s), \\ 0 & \text{otherwise;} \end{cases}$$

$$\gamma^F(s,a) = \{s' \mid Pr^F(s,a,s') > 0\}$$

$$\mathsf{succ}^F(s) = \bigcup \{s' \in \gamma^F(s,a) \mid a \in \mathsf{app}^F(s)\};$$

$$S^F = \text{transitive closure of } \mathsf{succ}^F \text{ over } S_0;$$

$$G^F = G \cap S^F;$$

$$\Sigma^F = (S^F, A, \gamma^F, \alpha, Pr^F, C);$$

$$P^F = (\Sigma^F, S_0, G^F).$$

Recall that in every place where the algorithm Z uses $\mathsf{app}(s)$, the algorithm Z^F instead uses $\{a \in \mathsf{app}(s) \mid \mathsf{acceptable}_F(s, det(a), \chi) \text{ holds}\}$. Thus from the above definitions, it follows that running Z^F on the planning problem P is equivalent to running Z on P^F.

The search-control function $\mathsf{acceptable}_F$ is *admissible* for P if for every state s in P, there is an action $a \in \mathsf{app}(s)$ such that $\mathsf{acceptable}_F(s, det(a), \chi)$ holds and we have

$$V(s) = C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s')V(s').$$

From the admissibility of $\mathsf{acceptable}_F$, the theorem below follows:

Theorem 13 *Suppose Z returns a solution policy π for P. Then, Z^F returns a solution policy π' for P such that $V_\pi(s) = V_{\pi'}(s)$ for every $s \in S_0$, if $\mathsf{acceptable}_F$ is admissible for P.*

Intuitively, the above theorem states that if the search-control function $\mathsf{acceptable}_F$ is admissible for the planning problem P then the modified planning algorithm Z^F never prunes an action that can be a part of an optimal solution policy for P. The proof of this theorem is given in the Appendix A.3.

Next, we consider the computational complexity of Z and Z^F. This depends heavily on the search space. If $P = (\Sigma, S_0, G)$ is a planning problem over an MDP $\Sigma = (S, A, \gamma, \alpha, Pr, C)$, then we define the *reachability graph* for P as the digraph $\Gamma_P = (N_P, E_P)$, where N_P is the transitive closure of succ over the initial states S_0 of P, and $E_P = \{(s, s') \mid s \in N_P, s' \in \mathsf{succ}(s)\}$. The Forward-VI algorithm searches the entire reachability graph of the planning problem P in order to generate a solution. For algorithms like RTDP and LRTDP, the search space is a subgraph of Γ_P.

The *depth* of the reachability graph Γ_P of P is the maximal distance between a state $s \in S_0$ and any other state in N_P. The *breadth* of Γ_P is $\max\{|\mathsf{succ}(s)| \mid s \in N_P\}$.

If F is a forward-chaining planning algorithm that is an instance of FCP, then the reachability graph for $\mathsf{Forward\text{-}VI}^F$ is $\Gamma_P^F = (S^F, E^F)$, where S^F is as defined earlier, and $E^F = \{(s, s') \mid s \in S^F, s' \in \mathsf{succ}^F(s)\}$. Then, we have the following: the ratio between the running times of Z and Z^F on P is $O(\frac{b^d}{(b^F)^{(d^F)}})$, where d and b are the depth and the breadth of Γ_P, respectively, and d^F and b^F are the depth and the breadth of Γ_P^F.

Note that d is always an upperbound on d^F (i.e, $d^F \leq d$) and b is always an upperbound on b^F (i.e., $b^F \leq b$), since the search-control function acceptable does not introduce any new actions. The worst case is where $b = b^F$ and $d = d^F$ (i.e., $\Gamma_{\mathcal{P}} = \Gamma_{\mathcal{P}}^F$), and therefore, the ratio between the running times of Z and Z^F is 1; this happens if F's search-control function, $\mathsf{acceptable}_F$, does not prune any actions from the search space.

On the other hand, there are many planning problems in which $\mathsf{acceptable}_F$ will remove a large number of applicable actions at each state in the search space (some examples occur in the next section). In such cases, we have $b^F << b$, and this can produce an exponential speedup, as illustrated in the following simple example. Consider the Forward-VI algorithm. Suppose Γ_P is a tree in which every state at depth d is a goal state, and for every state of depth $< d$, there are exactly b applicable actions and each of those actions has exactly k possible outcomes — i.e., the breadth of Γ_P is bk. Thus, the number of nodes in Γ_P is $\Theta((bk)^d)$. Next, suppose F's search-control function eliminates exactly half of the actions at each state. Then Γ_P^F is a tree of depth d and breadth $(b/2)k$, so it contains $\Theta(((b/2)k)^d)$ nodes. In this case, the ratio between the number of nodes visited by Forward-VI and Forward-VI^F is 2^d, so Forward-VI^F is exponentially faster than the original one.

5.4 Experimental Evaluation

This section presents an experimental evaluation of the modification technique described in the previous sections. The experiments are performed using Forward-VI, RTDP, and LRTDP, and their enhanced versions Forward-VI$^{\mathsf{SHOP2}}$, RTDP$^{\mathsf{SHOP2}}$, and LRTDP$^{\mathsf{SHOP2}}$. I implemented all six of the planners in LISP. [2]

For meaningful tests of the enhanced algorithms, these experiments involved planning problems with much bigger state spaces than in prior published tests of RTDP and

[2]The authors of RTDP and LRTDP were willing to let us use their C++ implementations, but we needed LISP in order to use SHOP2's search-control mechanism and to avoid integration issues between C++ and LISP parts of the the implementations of the modified planners that would arise otherwise.

LRTDP [BG03], where the biggest experimental problems in size had 383,950 states in their state spaces. For this purpose, I used the following two planning domains; which are MDP adaptations of Blocks World and Robot Navigation. The planning problems in the original versions of both of the Blocks World and Robot Navigation domains contain more than 50 million states in their state spaces, and so do their MDP adaptations.

All of the experiments were run on a AMD Duron 900MHz laptop with 256MB memory, running Fedora Core 2 Linux. In either of the experimental domains, each action had a unit cost of 1. The discount factor was $\alpha = 1.0$, as in the experiments with RTDP reported in [BG03], and the termination criterion was $\epsilon = 10^{-8}$ for all of the experimental planners. The modified planners were provided with the same input search-control information encoded as hierarchical task networks, and all six planners have been provided as input the same planning operator descriptions (i.e., action descriptions) in these experiments. All of the domain and problem descriptions used in these experiments will be accessible via http://www.cs.umd.edu/users/ukuter/mdps/.

The RTDP and LRTDP algorithms use domain-independent heuristics to initialize their value functions. I used two such heuristics. The first one, h_0, initializes the value of every state to 0. The other is the h_{min} heuristic reported in [BG03]:

$$Q(s,a) \leftarrow C(s,a) + \min_{s' \in \gamma(s,a)} Pr(s,a,s')\, V(s') \tag{5.1}$$

5.4.1 Probabilistic Blocks World

One of the experimental domains was the Probabilistic Blocks World (PBW) from the 2004 International Probabilistic Planning Competition [LY04]. This domain is a vari-

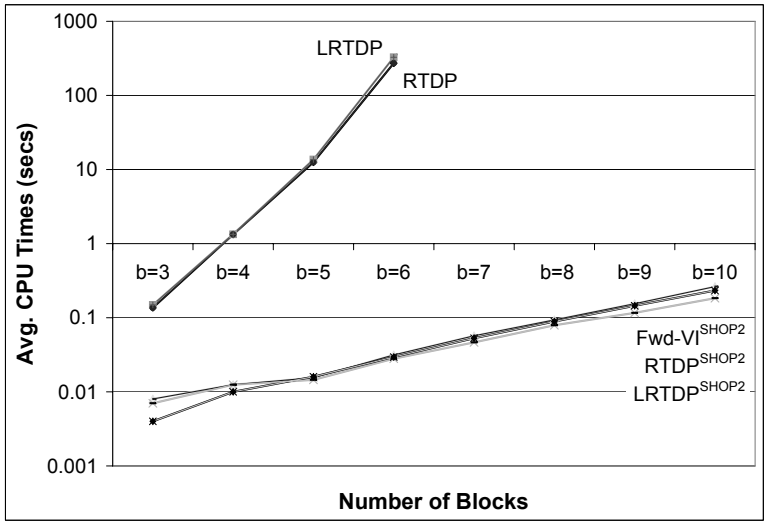

Figure 5.4: Running times for PBW using h_0, plotted on a semi-log scale. With 6 blocks ($b = 6$), the modified algorithms are about 10,000 times as fast as the original ones. Each data point is the average of 20 problems.

ation of the original Blocks World domain discussed in Section 2.2. As in the original version, there are four kinds of actions; namely, the pickup, putdown , stack, and unstack actions. However, each action may drop the block on the table with a 15% probability.

In Blocks World domain, the size of the state space in this domain grows combinatorially with the number of blocks: with 3 blocks there are only 13 states, but with 10 blocks there are 58,941,091 states. This is also true for the Probabilistic Blocks World since this probabilistic adaption of Blocks World has the same set of states as the original one.

On most of the problems, Forward-VI failed due to memory overflows, so there are no results reported for this algorithm in the following discussion. Figures 5.4 and 5.5 show the average running times of all the other five planners in the PBW domain, using the h_0 and h_{min} heuristics respectively. Each data point is the average of 20 runs. The running times for RTDP and LRTDP were almost the same, and so were those of the

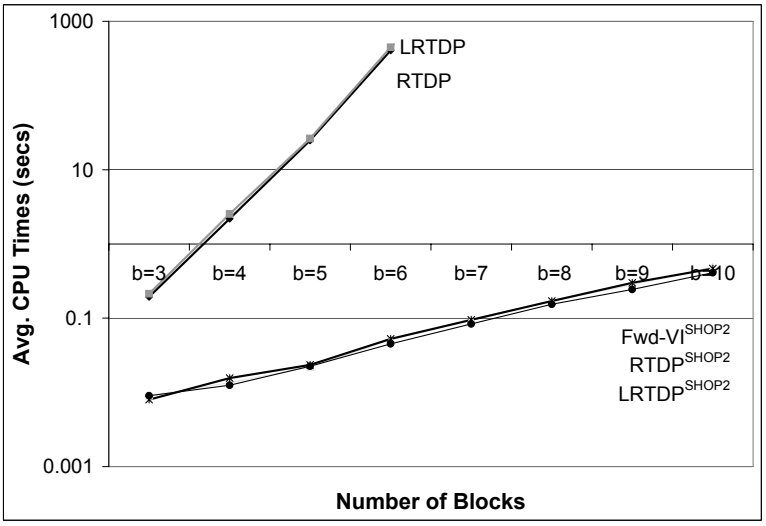

Figure 5.5: Running times for PBW using h_{min}, plotted on a semi-log scale. Like before, when $b = 6$ the modified algorithms are about 10,000 times as fast as the original ones. Each data point is the average of 20 problems.

three enhanced planners. Every algorithm's running time grows exponentially, but the growth rate is much smaller for the enhanced algorithms than for the original ones—for example, at $b = 6$ they have about 1/10,000 of the running time of the original algorithms. Once we got above 6 blocks (4,051 states in the state space), the original algorithms ran out of memory.[3] In contrast, the modified algorithms could easily have handled problems with more than 10 blocks (more than 58 million states).

The reason for the fast performance of the modified algorithms is that it is very easy to specify domain-specific (but problem-independent) strategies encoded in HTNs such as "if there is a clear block that you can move to a place where it will never need

[3]Each time RTDP or LRTDP had a memory overflow, they were run again on another problem of the same size. Each data point on which there were more than five memory overflows is omitted as before, but those data points where it happened 1 to 5 times are included in the results. Thus our data make the performance of RTDP and LRTDP look better than it really was—but this makes little difference since they performed so much worse than RTDP^{SHOP2} and RTDP^{SHOP2}.

to be moved again, then do so without considering any other actions," and "if you drop a block on the table, then pick it up again immediately." Such strategies reduce the size of the search space tremendously as already discussed for ND-SHOP2 and $\text{FS}^3_{\text{SHOP2}}$ in the previous chapters.

5.4.2 Probabilistic Robot Navigation

The second experimental domain was an MDP adaptation of the Robot Navigation domain [KBSD97, PBT01] described in Section 3.6. In this adapted domain, there is a building has 8 rooms and 7 doors. Some of the doors are called *kid doors*. Whenever a kid door is open, a "kid" can close it randomly with a probability of 0.5. If the robot tries to open a closed kid door, this action may fail with a probability of 0.5 because the kid immediately closes the door. Packages are distributed throughout the rooms, and need to be taken to other rooms. The robot can carry only one package at a time.

In the Robot Navigation domain and in the MDP adaptation of it described above, the state space contains 54,525,952 states, when there are 5 packages in the domain.

Tables 5.1 and 5.2 show the running times for the planners in the Robot Navigation domain. The times for RTDP and LRTDP grew quite rapidly, and they were unable to solve many of the problems at all because of memory overflows. $\text{RTDP}^{\text{SHOP2}}$ and $\text{LRTDP}^{\text{SHOP2}}$ had no memory problems, and their running times were quite small.

To try to alleviate RTDP's and LRTDP's memory problems, I created a Simplified Robot Navigation domain in which there are no kid doors. When the robot tries to open a door, it may fail with probability 0.1, but once the door is open it remains open.

Table 5.1: Running times using h_0 on Robot-Navigation problems with one kid door. p is the number of packages. Each data point is the average of 20 problems.

$p =$	1	2	3	4	5
RTDP	10.21	254.64	-	-	-
LRTDP	11.80	1622.68	-	-	-
Fwd-VI$^{\text{SHOP2}}$	0.01	0.03	0.06	0.08	0.14
RTDP$^{\text{SHOP2}}$	0.01	0.02	0.05	0.07	0.11
LRTDP$^{\text{SHOP2}}$	0.02	0.03	0.06	0.09	0.17

Table 5.2: Running times using h_{min} on Robot-Navigation problems with one kid door. p is the number of packages. Each data point is the average of 20 problems.

$p =$	1	2	3	4	5
RTDP	23.85	629.46	-	-	-
LRTDP	15.08	383.17	-	-	-
Fwd-VI$^{\text{SHOP2}}$	0.01	0.03	0.09	0.14	0.25
RTDP$^{\text{SHOP2}}$	0.01	0.03	0.08	0.13	0.22
LRTDP$^{\text{SHOP2}}$	0.01	0.04	0.09	0.14	0.26

Table 5.3 shows the results on this simplified domain using h_{min}. RTDP and LRTDP took less time than before, but still had memory overflows. As before, RTDP$^{\text{SHOP2}}$ and LRTDP$^{\text{SHOP2}}$ had no memory problems and had very small running times.

An explanation about the performance of RTDP and LRTDP in these experiments is in order. LRTDP is an extension of RTDP that uses a labeling mechanism to mark states whose values have converged so that the algorithm does not visit them again during the search process. In most our problems, we observed that the value of a state did not converge until towards the end of the planning process. As a result, labeling states did not

Table 5.3: Running times using h_{min} on Simplified Robot-Navigation problems, with no kid doors. p is the number of packages. Each data point is the average of 20 problems.

$p =$	1	2	3	4	5
RTDP	4.69	211.22	-	-	-
LRTDP	4.66	209.87	-	-	-
Fwd-VI$^{\text{SHOP2}}$	0.01	0.03	0.05	0.09	0.15
RTDP$^{\text{SHOP2}}$	0.01	0.02	0.04	0.08	0.14
LRTDP$^{\text{SHOP2}}$	0.01	0.03	0.05	0.09	0.15

really help improve the performance of LRTDP on those problems. In each such case, LRTDP spent a significant portion of its running time to unsuccessfully attempt to label the states it visited. As a result, RTDP was able to perform better than LRTDP in those problems since it is free from such overhead.

Chapter 6

Related Work

This chapter overviews the previous works on planning under uncertainty, with a focus on the most directly related ones to the contributions of this dissertation, and highlights the differences between those works and the ideas discussed here.

6.1 Planning in MDPs

In planning under uncertainty, the predominant approach is based on MDPs; see [BDH99] for an excellent survey of this approach. In MDP planning problems, actions have nondeterministic outcomes and the planner knows how likely each outcome will occur when an action is executed in the world. The objective is to find a policy (i.e., a plan expressed as a function that tells which action to perform in each state) that optimizes a utility function.

The primary approach to solve MDP planning problems is dynamic programming [Ber05]. Dynamic programming is general solution method for problems that involve making a sequence of optimal and/or near-optimal decisions [HS78]. The basis of dynamic programming is the well-known *Principle of Optimality*, which states that

> An optimal sequence of decisions has the property that whatever the initial state and the initial decision are, the rest of the decisions must consitute an optimal decision sequence with respect to the state resulting from the first

decision.

Dynamic programming techniques implement this principle by enumerating decision sequences that are relevant to an optimal solution to the input decision problem, avoiding those sequences that cannot be possibly optimal.

The two basic algorithms for solving MDPs are Value Iteration and Policy Iteration [Ber05]. The Value Iteration algorithm is a dynamic-programming technique that explores the state space until it converges to a policy of optimal expected utility. The Bellman Equation [Bel57], which is described in Section 2.3.1, is the basis of the Value Iteration algorithm. Value Iteration is simply an iteration loop: at each iteration, the algorithm considers every state in the state space and updates its value by the Bellman Equation. The forward version of Value Iteration, one of the algorithms that this dissertation focused on in Chapter 5, does not update the value of every state in the state space, but it limits its computation only to those states that are reachable from the initial states of the input MDP planning problem. The rationale behind this approach is that a solution for an MDP planning problem in our setting must reach to the goal states with probability 1, and therefore, there is no point in considering a state that is not reachable from an initial state since that state does not have any effect on the optimal solution for the planning problem.

One issue in Value Iteration is that the value updates are not linear; in order to guarantee convergence to the optimal value for each state in the state space, the algorithm must be run infinitely often. In practice, Value Iteration is used to generate near-optimal solutions by defining a termination condition ϵ, where the algorithm is terminated when

an iteration updates the values of every state by no more than ϵ.

The Policy Iteration algorithm is similar to Value Iteration in that it also uses the Bellman Equation in order to update the values of the states in the state space. The primary difference is that Policy Iteration iterates over the space of all possible policies, rather than the state space. The algorithm alternates between a policy evalution phase and a policy update phase. In the former, the values of the states in the current policy is updated using the Bellman Equation. In that latter phase, the algorithm examines each state and the action specified for that state by the current policy and compares the value of applying that state-action pair with that of other possible actions applicable in that state. If there is a better action for that state, then the algorithm updates the current policy with that action. The process continues until no further policy improvement is possible. The Policy Iteration algorithm always terminates after a finite number of iterations, since there are only finitely many possible policies.

Despite its general applicability and mathematical soundness, the problem of finding optimal policies in MDPs is often computationally impractical due to the typical requirement of enumerating the entire state and/or action spaces. In fact, it has been theoretically shown that for planning problems expressed using probabilistic STRIPS operators [HM93, KHW94] or 2TBNs [HSAHB99, BG96], MDP planning is EXPTIME-hard [Lit97].

Several approaches have been developed to address the complexity of MDP planning. In one venue, there has been attempts to extend classical planning to MDP planning. For example, [KHW94] described an *plan assessment-and-generation* approach in which goals are sets of states, and the planning problem is to generate partially-ordered plans by

requiring them to reach to a goal state with probability over a given threshold given a set of initial states. This approach suffers from an significant drawback: as plan-space planners search in a space of partial plans; doing so, they lose the state information that would otherwise be generated during planning as in state-space search, which usually allows for using heuristics to guide the planning process.

As another example, [BL99] described PGP, a straightforward generalization of GraphPlan [BF97] for MDP planning. In PGP, a planning graph is constructed by a forward search starting from the initial states of an MDP planning problem, just like in the original GraphPlan algorithm. Then, this planning graph is used to guide a dynamic programming step, which computes the values of the policies represented by the planning graph. The planning graph is successively extended until the dynamic programming step cannot generate a better policy.

One idea was to use *abstraction and aggregation* techniques in order to reduce the size of the state space that needs to be explored by MDP planning algorithms. Examples of this approach include [DB97, LK02, GK03, DKKN95, HS94]. The DRIPS planning system [HS94] is one of the earliest attempts for problem abstraction in MDP planning. Here, the abstraction is achieved by the use of *macro operators*, i.e., high-level planning operators that specify a sequence of primitive actions together such that applying a macro operator in a state have the same outcome as applying the sequence of actions that it represents. This approach allows to search in an abstract state space to generate solutions for MDP planning problems. Macro operators have been usually, and wrongly, confused with Hierarchical Task Networks (HTNs). HTNs are strictly more expressive than macro operators since they provide multiple levels of abstraction, rather than a single

level. Furthermore, the hierarchical abstraction in HTNs are computed on-the-fly during the planning process, whereas macro operators need to be fixed in the input of a planning algorithm.

In [DKKN95], the notion of using *envelopes* has been introduced: an envelope is a smaller portion of the state space that abstracts away from the states that are not relevant to an optimal solution. The algorithms based on this technique starts with an initial envelope and incrementally extend it until a solution is found. [GK03] described an application of this idea, where the classical planning algorithm GraphPlan [BF97] is used for fast generation of the initial envelope. The authors have demonstrated the applicability of their approach in simple planning problems, and discussed its scability in larger problems.

One of the first attempts for using state aggregation in MDPs was introduced in [DB97]. In this work, states are clustered together to form an abstract version of the input MDP. This technique is based on the abstraction method described in [Kno94] for classical planning that ignores some of the literals from the description of a classical planning problem. In the MDP context, those literals that have little impact on the value of an optimal policy are ignored. The abstract MDP constructed in this way has the advantage over the envelope-based approaches that none of the states in state space is actually ignored during planning but only the irrelevant portions of those states. However, the solutions generated by this approach are approximate. However, the authors describe how to compute bounds over the divergence from the optimal solution and near-optimal policies can be compared by using these bounds.

The notion of *factorized MDPs* has been introduced based on the observation that in

many planning problems, the state space often admits a factored representation [BDH99], i.e., each state can be represented as a collection of *state variables*. In planning problems where this is true, often the utility of applying an action in a state will not depend on all of the state variables that describe that state, but only on some of them. One particularly popular factorization approach is to use linear representations of the value function associated with an MDP [Ber05, KP99, KP00]. In this approach, the original value function is represented a linear combinations of a number of basis functions, which are defined by extracting the features of the MDP [TVR96]. The coefficients of those basis functions in a particular linear form are computed by approximate linear programming techniques [dFVR03, TZ97, GKP01b, SP01].

Factored representations of MDPs have been also fruitful in the use of *decision trees and diagrams* [HSAHB99, BDG00] and other symbolic approaches [BRP01, FH02] for MDP planning. In [HSAHB99], Algebraic Decision Diagrams (ADDs) are used to represent the reward functions, costs, probabilistic state-transitions, and the value functions in factorized MDPs.[1] [FH02] has extended this approach to generalize the heuristic search algorithms LAO* [HZ98] and RTDP [BG00] to work over ADD-based representations of MDP planning problems.

The LAO* planning algorithm [HZ98, HZ01] is a generalization of AO*, the well known algorithm for searching AND-OR graphs [Nil80], for dealing with cycles in those graphs. LAO* is a heuristic search algorithm that consists of an iteration loop in which the algorithm performs a forward search starting from the initial states toward the goal

[1] ADDs are generalizations of Binary Decision Diagrams, which were the focus of a part of this dissertation, where a number is associated with a boolean formula, rather than a truth value.

states of the input MDP planning problem. This forward search is followed by a dynamic programming step in which the value of each state that is in that policy or that is an ancestor of a state in that policy in the state space is updated. The iteration continues until the difference between the values of the states in the successive best policies falls below an ϵ error threshold.

Real-Time Dynamic Programming was first introduced by [BBS95] as a technique for solving for Reinforcement Learning [SB98] problems, and later adapted for planning under uncertainty by [BG00, BG03] as described in this dissertation. RTDP is a combination of the ideas from *real-time search* [Kor90], *greedy search* [RN03], *stochastic sampling* [BT96], and dynamic programming in a single unified framework for MDP planning. RTDP performs forward search by simulating possible executions of the actions in the states generated during this search. Each forward search starts from the initial state of the input MDP planning problem and ends in a goal state. The algorithm performs successive forward searches of this sort until some convergence criterion is met.

Simulation based MDP planning has been extensively used in reinforcement learning [SB98, Wat89] and systems control [Mac66, EDMM03, CFHM05]. Reinforcement learning is a form of planning under uncertainty where the planning (i.e., learning) process is usually characterized as trial-and-error search and the objective is to generate a policy that optimizes some utility value. Despite the similarities between traditional MDP planning techniques and reinforcement learning, one important difference is that the latter does not require the model of the underlying planning domain to be known; instead, reinforcement learners, such as the TD(λ)-family of algorithms and SARSA [SB98], and Q-Learning [Wat89], can discover it by trying actions in states and observing the out-

comes. Due to this property, most researchers believe that reinforcement learning is very suitable for robotic applications, systems control, games, and other applications.

In systems control, a recent simulation based algorithm is AMS [CFHM05], which is designed for finite-horizon MDPs with large state spaces but relatively smaller action spaces. AMS can be seen as a search of a decision tree, where each node of the tree represents a state (with the root node corresponding to the initial state) and each edge signifies a sampling of a given action. It employs a depth first search for generating sample paths from the initial state to the final state (i.e., when a given finite horizon is reached) and uses backtracking to estimate the value functions at visited states. The algorithm uses the ideas from multi-armed bandit problems to adaptively sample applicable actions during the search process.

One particular simulation-based approach is the idea of *action elimination* during planning. The notion of action elimination was originated by MacQueen[Mac66], where some inequality forms of Bellman's Equation are used together with bounds on the optimal value function to identify and eliminate non-optimal actions in order to reduce the size of the action spaces to be searched. Since then, action elimination has been applied to several standard MDP solution techniques such as value iteration and policy iteration, see e.g., [Put94] for a review. In a recent work [EDMM03], the idea of action elimination has been explored in a reinforcement learning context where the explicit mathematical model of the underlying system is unknown.

The key idea in all the aforementioned approaches is to avoid enumerating and searching the entire state space in MDP planning problems. However, this approaches are general and domain-independent planning algorithms that does not exploit any informa-

tion on the input MDP. Some of the new planning algorithms that have been described in this dissertation, on the other hand, are domain-independent search engines with the ability to use domain-specific search-control information. This property makes these algorithms solve larger problems very efficiently, as demonstrated in the experimental evaluations in Section 5.4.

MDP planning has been extended to the partial-observability case via Partially-Observable MDPs (or POMDPs, for short) [Son78, BG00, CKL94, KLC98, PB00, PB01, Kar01, BDH99]. POMDP planning can be seen as searching over a space of belief states. In its general form, a belief state is defined by a probability distribution over its member states, and planning is done via transformations of these probability distributions. However, this formulation makes the POMDP planning very hard: the number of belief states is huge and in most cases, it may not be finite. As a result, POMDP planning algorithms can only solve very simple and toy planning problems, and they cannot scale up to complex ones. Among the few planning algorithms that have demonstrated some practicality are GPT [BG01a] and PTLplan [Kar01].

6.2 Planning in Nondeterministic Domains

This dissertation is not the first attempt to extend classical planning algorithms for planning with under nondeterminism. Probably the first work in a similar vein is described in [GN93], which is a breadth-first search algorithm over an AND-OR tree. Other early attempts to extend classical planning to nondeterministic domains are *conditional planning* techinques, including the CASSANDRA planning system [PC96], CNLP [PS92], PLINTH

[GB94], UCPOP [PW92], and MAHINUR [OP99]. Unfortunately, these extensions of classical planning do not perform well in many planning problems and they cannot scale up to complex planning domains. Furthermore, conditional planning techniques do not address the problem of infinite paths and of generating trial-and-error strategies.

The PKS planning algorithm [PB02] introduced a different formulation of planning problems than the previous approaches. In PKS, planning problems under nondeterminism are modeled using a "knowledge-level formulation," where the planner does not reason over what is actually true or false about in the world, but instead, it reasons over what it knows to be true or false about the states and the outcomes of its actions. PKS generates conditional plans with sensing actions, which does not change the world state but provide information about it. Planning in PKS is done by interleaving the executions of those sensing actions and the actual actions that change the world state.

[Rin99a] introduced QBFPLAN, another novel planning algorithm that is a generalization of satisfiability-based planner SATPlan [KS92]. QBFPLAN translates a nondeterministic planning problem into a satisfibility problem over *Quantified Boolean Formulas (QBFs)*. The QBF problem is then fed to an efficient QBF solver such as the one described in [Rin99b]. QBFPLAN generates conditional plans that are bounded in length by a parameter specified as input. If a solution cannot be found within the current length, then the algorithm extends this bound and starts all over again. As its satisfiability based predecessors, QBFPLAN does not seem to scale up to large planning problems.

One of the earliest attempts to use model checking techniques for planning under nondeterminism was first introduced in [KBSD97]. SIMPLAN, the planning system described in [KBSD97], is developed for generating plans in reactive environment, where

such plans specify the possible reactions of the world with respect to the actions of the plan. Note that such reactions can be modeled as nondeterministic outcomes of the actions. SIMPLAN models the interactions between the environment and the execution of a plan by using a state-transition system, which specifies the possible evolutions of the environment due to such interactions. Goals over the possible evolutions of the enviroment are specified by using *Linear Temporal Logics (LTL)* as in the classical TLPlan algorithm [BK00], which, in fact, takes its roots from SIMPLAN.

The SIMPLAN planner is based on model checking techniques that work over explicit representations of states in the state space; i.e., the planner represents and reasons explicitly about every state visited during the search. Symbolic model-checking techniques, such as Binary Decision Diagrams [Bry92], to do planning in nondeterministic domains under the assumptions of fully-observability and classical reachability goals was first introduced in [CGGT97, GT99]. BDDs enable a planner to represent a class of states that share some common properties and the planning is done by transformations over BDD-based representations of those states. In some cases, this approach can provide exponential reduction in the size of the representations of planning problems, and therefore, exponential reduction in the times required those problems, as both demonstrated in this dissertation and in previous works [CPRT03, PBT01].

The planning algorithms developed within this approach aim to generate solutions in nondeterministic planning domains that are are classified as weak (at least one execution trace will reach a goal), strong (all execution traces will reach goals), and strong-cyclic (all "fair" execution traces will reach goals) [CRT98a, CRT98b, DTV99]. [CPRT03] gives a full formal account and an extensive experimental evaluation of plan-

ning for these three kinds of solutions.

Planning as model checking has been extended to deal with partial observability [BCRT01, BCRT06]. In these works, belief states are defined as a classes of states that represent common observations, and compactly implemented by using Binary Decision Diagrams [Bry92]. Planning is done by performing a heuristic search over an AND-OR graph that represents the belief-state space. It has been demonstrated in [BCRT06] that this approach outperformed two other planning algorithms developed for nondeterministic domains with partial observability; namely GPT [BG01a] and BBSP [Rin05].

Planning with complex goals in nondeterministic planning domains has been also investigated in several works, including [BCRT06, PT01, PBT01, DPT02]. The MBP planner [BCP$^+$01] that is used as a benchmark in the experimental evaluations described in this dissertation is capable of handling both.

Other planning algorithms that are based on model checking techniques include the UMOP planner, described in [JV00, JVB01, JVB03] is a symbolic model-checking based planning framework and a novel algorithm for strong and strong-cyclic planning which performs heuristic search based on BDDs in nondeterministic domains [JVB03]. Heuristic search provides a performance improvement over the unguided BDD-based planning techniques on some toy examples (as demonstrated in [JVB03]), but the authors also discuss how the approach would scale up to real-world planning problems.

As another example, [GMP00, GPM99] describes a model-checking approach to planning that uses timed automata to guarantee that certain timing constraints are met in the generated plans. CIRCA, a planning system that has been developed using this approach, performs a forward state-space search in a nondeterministic planning domain

starting from a set of initial states in order to generate a policy that achieves the goals. To prevent the exponential blow-up that can occur in a forward search, a domain-independent heuristic based on computing a lookahead in order to choose the best action to apply in a state is used as a part of the planner. [YMS03] generalized this technique to perform stochastic sampling to generate solutions for MDP planning problems. Although this approach has demonstrated to be successful for the applications in which CIRCA was exploited, in the general case, the number of samples required to assess acceptable estimations for the probability values may be exponential in the size of the state space, which could be huge (even infinite) in most of the real-world planning problems.

Finally, several other approaches have been developed for planning under nondeterminism, mostly focusing on conditional and conformant planning. These approaches extend classical planning techniques based on planning graphs [BF97] and satisfiability [KS92]. Satisfiability based approaches, such as the ones described in [CGT03, FG00, Giu00], are limited to only *conformant planning*, where the planner has nondeterministic actions and no observability. The planning-graph based techniques [SW98, WAS98, BK04] can address both conformant planning and a limited form of partial observability.

Chapter 7

Closing Remarks

7.1 Conclusions

This dissertation have described a suite of new approaches that have produced new planning algorithms for planning under uncertainty. Some of the new planning algorithms are developed for nondeterministic planning domains and others for MDP planning problems. In both settings, actions may have more than one possible outcomes and the planner does not know which outcome will actually occur when an action is executed in the world. In MDPs, the planner knows how likely each outcome may occur, whereas in nondeterministic planning domains, this information is not available. The objective of MDP planning is to generate a plan that optimizes some expected utility when executed, whereas in nondeterministic domains, the objective is to generate a plan whose execution must achieve a condition (either during the execution or at the state where it is ended).

Planning in nondeterministic planning domains and in MDPs violates most of the restrictive assumptions made in classical planning, rendering classical planning algorithms inapplicable verbatim in such domains. However, some of these classical techniques can be generalized to work in nondeterministic planning domains and in MDPs, yielding new algorithms for planning under uncertainty, which are more efficient than previous ones. Such generalization approaches were the focus of this dissertation.

The first technique described here was a method to take any forward-chaining classical planning algorithm, and systematically generalize it to work in nondeterministic planning domains. There are significant classes of nondeterministic planning problems in which the number of possible states is exponential but the sizes of the solutions are polynomial in the size of those problems. This dissertation has presented theoretical results that suggest that in such domains, nondeterminizations of efficient classical planners may be able to do very well. The theoretical results are confirmed by an experimental evaluation and complexity analyses on two different problem domains. In the experiments, ND-SHOP2, a "nondeterminization" of SHOP2 [NAI+03] produced by our planner-generalization method, could find solutions in nondeterministic planning domains about two to three orders of magnitude faster than MBP [BCP+01], which was one of the best previous planners for such domains.

The primary reason that the generalized planning algorithms can generate solutions very efficiently in nondeterministic planning domains is that they can use effective search-control information to focus their search for solution plans. However, when effective search-control information is not available due to the complexity of a nondeterministic domain, the efficiency of the generalized planning algorithms usually degrades substantially.

Forward State-Space Splitting (FS^3), the second planning technique described here, has been aimed to combine the ability of using search-control information of the generalized planners with symbolic model-checking techniques in order to reason during planning about classes of states that share some common properties. In particular, FS^3 allows to take the pruning technique of any forward-chaining classical planner, such as TLPlan,

TALplanner, and SHOP2, and use it in planning via Binary Decision Diagrams (BDDs). The theoretical results presented in this dissertation described the correctness, the completeness, and the termination properties of this approach. In the experiments, $\text{FS}^3_{\text{SHOP2}}$, one of the new algorithms that combines hierarchical task networks as in SHOP2 with BDDs as in MBP, was never dominated by either MBP or ND-SHOP2: $\text{FS}^3_{\text{SHOP2}}$ could easily deal with problem sizes that neither the other two approaches could scale up to, and it could solve problems about two or three orders of magnitude faster than the other two.

As the third and the final approach, this dissertation have described a way to incorporate the search-control mechanism of classical planners, such as SHOP2, TLPlan, and TALplanner, into the previous planning algorithms originally developed for MDPs. If the search-control function of the original classical planning algorithm satisfies an "admissibility" condition, then the modified MDP planner is guaranteed to find optimal solutions. If the search-control algorithm generates a smaller set of actions at each state than the original MDP algorithm did, then the modified planner will run exponentially faster than the original one. In an experimental evaluation of this approach where the search-control algorithm from SHOP2 has been incorporated into three MDP planners RTDP [BG00], LRTDP [BG03], and Forward-VI, a forward-chaining version of Value Iteration [Ber05], the enhanced algorithms were about 10,000 times faster than the original ones on the largest problems the original ones could solve. On another set of problems that were more than 14,000 times larger than the original algorithms could solve, the enhanced ones took only about 1/3 second.

7.2 Future Work

The techniques described in this dissertation have good potential to be applicable in other research areas as well. This section highlights some of those research areas, namely Reinforcement Learning, Hybrid Systems Control, and Planning under Temporal Uncertainty, and discusses how the ideas of this dissertation can be generalized to work for problems in those fields.

7.2.1 Reinforcement Learning

Reinforcement learning can be seen as a form of planning under uncertainty in MDPs, where the planning (i.e., learning) process is usually characterized as trial-and-error search and the objective is to generate a policy that optimizes some utility value [SB98]. Despite the similarities between MDPs and reinforcement learning, one important difference is that the latter does not require the model of the underlying planning domain to be known; instead, reinforcement learners can discover it by trying actions in states and observing the outcomes. Due to this property, most researchers believe that reinforcement learning is very suitable for robotic applications, systems control, and games.

Typical solution methods for reinforcement learning problems are based on dynamic programming and simulation techniques, and they usually require exploring all or most of the state space in order to generate optimal or near-optimal policies. In complex problems, the sizes of the state spaces are usually prohibitive as in MDP planning. To address this issue, Reinforcement Learning algorithms that use hierarchical abstract machines [Par98] and MAX-Q decompositions [Die00] have been developed. These tech-

niques are based on hierarchical abstractions that are somewhat similar to Hierarchical Task Networks (HTNs), and they are analogous to an instance of the decomposition tree that an HTN planner might generate. However, the abstractions must be supplied in advance by the user, rather than being generated on-the-fly as in an HTN planner.

In FS^3, much of the computational machinery was for correctly handling the possible state transitions induced by the nondeterministic actions — a characteristic that nondeterministic planning shares with MDP planning and Reinforcement Learning. This suggests that it should be possible to generalize FS^3 for solving MDP and Reinforcement Learning problems. This requires developing new techniques for handling the probabilities, rewards, and costs in the state-transition operations dictated by the transformations over the BDDs and the search-control strategies.

Once the new solution methods are complete, it will be possible to investigate the relationships between the search-control techniques developed in the fields of planning and reinforcement learning, such as planning with hierarchical task networks and the reinforcement learning techniques that exploit hierarchical abstractions. This will bring the two research fields closer and pave a way to further cross-fertilizations between them.

7.2.2 Hybrid Systems and Control

Hybrid systems are dynamical systems that have both continuous- and discrete-valued state variables [LTS99, TMBO03]. Examples of such systems include robotic systems, tactical fighter aircrafts, intelligent vehicle/highway systems, and flight control systems. The inherent uncertainties and interactions between the discrete and continuous

components make it very hard to synthesize optimal controllers for such systems.

Typical existing approaches for hybrid systems model the discrete and continuous components of a hybrid system independently and generate controllers using techniques that can exploit the interactions between the two separate models. Examples include the use of timed automata [AD94], game-theoretic approaches [TLS00], linear hybrid automata [SPS00], and linear programming and simulation techniques [HK04].

Combinations of BDD-based state representations and search-control strategies as in FS^3 can also be used in synthesizing controllers for hybrid systems in order to abstract away from the continuous parts of the state space, which is usually infinite due to the continuous-valued state variables in those systems. This approach primarily involves developing algorithms similar to FS^3 that decompose the system models into smaller and smaller models until a solution controller is generated. The results on FS^3 described in this dissertation suggests that this approach will compare favorably with the previous techniques for synthesizing controllers for hybrid systems.

7.2.3 Planning under Temporal Uncertainty

Many practical planning problems require reasoning about the temporal characteristics (e.g., durations) of the actions as well. In many cases, the execution of an action is not instantaneous; instead, there is a time interval between the start and the end of that execution. Even such action durations alone violates the requirements of classical planning. What is worse, however, is that a planner may not know in advance the exact duration that an action might take when it is executed. This requires a planner reason about the

possible durations for the actions, and generate plans that would guarantee the successful executions of those actions despite possible conflicts that may arise due to the times they are executed.

The planner-generalization method described in this dissertation can be extended to develop similar generalization methods that will provide new temporal-planning algorithms that will exploit those traditional search and pruning techniques. One of the approaches will involve using constraint-based representations. The expressive power of constraint-based representations has enabled their use in several practical settings—but the existing planners that use such representations have tended to be quite slow, except a few ones that are tuned for the specific planning domains in which they are intended to work. The new planning algorithms produced by the generalization methods will use effective pruning techniques to explore only the relevant portions of the search spaces, so it is likely that several of these algorithms will work quite efficiently. Theoretical and experimental investigations of this hypothesis is a research topic as a next step of this dissertation.

Once generalization methods for temporal planning, hybrid systems, and reinforcement learning have been developed, it will be possible to combine several of them, to produce new solution methods for planning and decision making that can handle both time and nondeterministic actions, or time and probabilistic actions. This work will provide new and efficient methods for planning and decision making with temporal uncertainty, nondeterminism, contingencies, continuous states, probabilities, and utilities. The results of this dissertation suggests that these solution methods will be able to solve larger classes of problems than the existing ones developed for such conditions do.

7.3 Challenges of Using Search Control under Uncertainty

Throughout this dissertation, it has been both theoretically and experimentally demonstrated that generalizing classical planning techniques that allow to use search-control information to guide planning is an effective approach for planning under uncertainty. Although, in principle, both domain-independent and domain-specific search-control heuristics can be used, it is not very realistic to expect that domain-independent search-control information would be proven to be effective except in very simple planning problems. This is because planners need to compute such search-control information for all or most states in the state space, and the state space of planning problems in uncertain domains are usually huge. The domain-specific search-control information, on the other hand, are compiled by domain experts and provided to the planners as an input. Some of the classical planning algorithms considered here use very expressive languages to specify domain-specific search-control information, and therefore, the pruning done based on such information is very effective. Forward planners are particularly successful in using such search-control information since they know the current state of the world all the time, and therefore, they can extract information from the states of the world for which action to choose in those states.

However, compiling domain-specific search-control information can be very difficult (or even impossible) in complex planning problems in uncertain environments. For example, there may be many possible outcomes of an action all of which may not be anticipated in advance. In some cases, all of the possible outcomes of an action may not be even known. Even if we know the exact possible outcomes of actions in a planning

domain, the decision of which action is to choose in a state may not only depend on that state but it may also depend on the future decisions to be made during planning. In all of such cases, it is simply not practical to expect from the domain experts to provide the search-control information to the planning algorithms.

One possibility to address this problem is to develop automated learning techniques to learn such search-control information. In classical planning, this approach has recently been started to being investigated, particularly in the form of learning hierarchical domain-specific knowledge. In [INMA06, INMAA05], the authors describe how to use concept learning algorithms from machine learning literature in order to learn domain knowledge in the form of hierarchical task networks as in SHOP2, ND-SHOP2, and FS^3. In another work, [LC06] describes a novel learning technique to learn hierarchical knowledge that is encoded in terms of a logical formalism. All of these learning approaches have been demonstrated to be effective in acquiring domain-specific knowledge, which later can be used as search-control information in the planning algorithms that can reason over the formalisms that these learning techniques use. However, these learning approaches only work in classical planning domains, and it is not straightforward to generalize these approaches to nondeterministic planning domains and MDPs.

Another possibility is the following. Although, under uncertainty, it is difficult to compile complete domain-specific search-control information that would guide a planning in most of the time, it is often possible to compile incomplete search-control information that can be used in some parts of the planning domains. For example, an incomplete search-control information may specify high-level strategies in a planning domain, whereas the low-level task of action selection needs to be done by the planners. In parts

of a planning domain for which the search-control information is available, planners may use that information for guiding the search, and in other parts, they may use domain-independent search control information (whenever it is easy to compute). Alternatively, hybrid planning systems can be developed; in this case, the planner consists of two planning algorithms, one has the ability to use search control and the other does not. The former algorithm is used whenever search-control information is available during planning, and the process switches to the latter whenever it is not. This approach would be particularly effective if the latter one can plan over compact state representation such as BDDs. For example, any instance of the FS^3 procedure may be combined with the MBP planner in order to produce such a hybrid planning system. This is a research direction that is being pursued as a follow-up work for this dissertation.

Appendix A

Proofs of the Theorems

A.1 Proofs for Chapter 3

Theorem 1 *Suppose one of the search traces of* ND-Λ *returns a policy* $\pi_{\chi'}$ *for* P' *given* χ'. *Then* $\pi_{\chi'}$ *is a solution policy for it.*

Proof. The proof is by contradiction and it is given in three parts, each for weak, strong, and strong-cyclic planning with ND-Λ.

Strong-Cyclic Planning. Suppose the planning problem P' is solvable given the search-control information χ'. Suppose one of the search traces of ND-Λ returns the policy $\pi_{\chi'}$ and suppose $\pi_{\chi'}$ is not a strong-cyclic solution for P'. According to the definition of strong-cyclic solutions, if $\pi_{\chi'}$ is not a strong-cyclic solution for P' then there exists at least one path, say p, in the execution structure $\Sigma_{\pi_{\chi'}}$ that does not start at an initial state and end at a goal state. The following shows that this is not possible.

First of all, note that every execution path in the execution structure $\Sigma_{\pi_{\chi'}}$ starts from an initial state of the planning problem P' since ND-Λ is a forward-chaining algorithm that starts from the initial states and explores only those states that are reachable from the initial states by applying actions successively. Therefore, if $\pi_{\chi'}$ is not a strong-cyclic solution then $\Sigma_{\pi_{\chi'}}$ contains an execution path p that does not end in a goal state. There are only two possible cases in which this can happen.

- p ends in a terminal state s that is not a goal state. This means that either there are no actions applicable in s or the search-control information χ' prunes away all of the applicable actions. In both cases, ND-Λ returns FAILURE rather than $\pi_{\chi'}$.

- p contains a cyclic sequence of states $\langle s_0, s_1, s_2, \ldots, s_k \rangle$ such that $s = s_k$ and there is no execution path starting from any s_i and ending at a goal state in $\Sigma_{\pi_{\chi'}}$. In other words, s has no $\pi_{\chi'}$-descendant in the set G of goal states. If this is the case, then, again, ND-Λ returns FAILURE rather than the policy $\pi_{\chi'}$, which is a contradiction.

Therefore, if one of the search traces returns a policy $\pi_{\chi'}$, then $\pi_{\chi'}$ is a strong-cyclic solution for the input planning problem P'.

Strong Planning. Suppose the planning problem P' is solvable given the search-control information χ'. Suppose one of the search traces of ND-Λ returns the policy $\pi_{\chi'}$ and suppose $\pi_{\chi'}$ is not a strong solution for P'. According to the definition of strong solutions, if $\pi_{\chi'}$ is not a strong solution for P' then there exists at least one path, say p, in the execution structure $\Sigma_{\pi_{\chi'}}$ such that either p contains a cycle or it does not start at an initial state and end at a goal state. The following shows that this is not possible.

Suppose p contains a cyclic sequence of states of the form $\langle s_0, s_1, s_2, \ldots, s_k \rangle$ such that $s = s_k$. In this case, ND-Λ returns FAILURE when it explores s_k, which is a contradiction. Now, suppose p it does not start at an initial state and end at a goal state. As in the proof for strong-cyclic case above, every execution path in $\Sigma_{\pi_{\chi'}}$, including p, starts from an initial state since ND-Λ is a forward-chaining algorithm. Then p must not end in a goal state, which means that p ends in a terminal state s in which there are no applicable actions given χ'. However, if this is the case, ND-Λ returns FAILURE rather than $\pi_{\chi'}$ as

described above, which is a contradiction.

Therefore, if one of the search traces returns a policy $\pi_{\chi'}$, then $\pi_{\chi'}$ is a strong-cyclic solution for the input planning problem P'.

Weak Planning. Suppose the planning problem P' is solvable given the search-control information χ'. Suppose one of the search traces of ND-Λ returns the policy $\pi_{\chi'}$ and suppose $\pi_{\chi'}$ is not a weak solution for P'. According to the definition of weak solutions, if $\pi_{\chi'}$ is not a weak solution for P' then, there is an initial state s_0, from which there does not exist a path in $\Sigma_{\pi_{\chi'}}$ that ends in a goal state. This means that every execution path in $\Sigma_{\pi_{\chi'}}$ that starts at s_0 either ends at a non-goal terminal state or induces a cycle such that there is no possibility of reaching to a goal state from any of the states in that cycle. In both cases, ND-Λ returns FAILURE rather than $\pi_{\chi'}$ as shown above: in the former, it returns FAILURE there is no actions applicable in the non-goal terminal state, and in the latter, the $\pi_{\chi'}$-descendancy check fails. Therefore, if ND-Λ returns a policy $\pi_{\chi'}$ then it is a weak solution for the input planning problem P'. ■

Theorem 2 *Suppose that $P' = (S_0, G, \Sigma)$ is χ'-solvable. Then, at least one of the search traces of ND-Λ returns a solution policy.*

Proof. Let the size of a solution policy $\pi_{\chi'}$ for P' be the number of state-action pairs in $\pi_{\chi'}$. Let the *minimum solution depth* for the planning problem P' be the minimum size of any solution for P'. The proof is by induction on n, the minimum solution depth for P':

Base Step ($n = 0$). In this case, $S_0 \subseteq G$ so ND-Λ inserts every state $s \in S_0$ into the $solved$ set and returns the empty policy.

Induction Step. Let $n > 1$ and suppose the theorem is true for every $k < n$. Let $S = \mathsf{StatesOf}(OPEN)$ and let (s, χ') be a pair in the $OPEN$ set; i.e., $s \in S$. Then, there must be an action a such that $\mathsf{acceptable}(s, a, \chi')$ is true and the planning problem $((S \setminus \{s\}) \cup \gamma(s, a), G, \Sigma)$ must have the minimum solution depth $n - 1$ given the search-control information $\mathsf{progress}(s, a, \chi')$ for every state in $\gamma(s, a)$. This is true since otherwise, the minimum depth of (S, G, Σ) could not be n. Then, one of the nondeterministic choices of ND-Λ will choose this action and recursively invoke itself with the input $((OPEN \setminus \{(s, \chi')\}) \cup \{(s', \mathsf{progress}(s, a, \chi')) \mid s' \in \gamma(s, a)\})$. By the induction hypothesis, this invocation computes a solution policy π' for the planning problem $((S \setminus \{s\}) \cup \gamma(s, a), G, \Sigma)$. Thus, ND-$\Lambda$ returns $\pi' \cup \pi \cup \{(s, a)\}$. ■

Lemma 3 *Suppose Λ returns a solution plan π for the classical planning problem P. Then, one of the search traces of ND-Λ also returns π for P.*

Proof. Since P is a classical planning problem, we have only deterministic actions in P; i.e., $|\gamma(s, a)| \leq 1$ for all states in P. In this case, ND-Λ reduces to Λ since (1) ND-Λ's $OPEN$ set always contains a single element at every invocation of the algorithm, and (2) the candidate solutions generated in each invocation of ND-Λ are plans, i.e., sequences of actions, rather than policies. Also, there are no π- descendants of a state visited in each invocation induced by the partial policy (i.e., plan) π.

Thus, ND-Λ has exactly the same search traces as Λ on the planning problem P, and one of those traces return π. ■

Theorem 4 *Suppose Λ finds solution plans in time $O(\rho(|\pi_\chi|))$ in a strongly-connected classical planning domain, given the search-control information χ. $|\pi_\chi|$ is the size of the solution plan and ρ is a monotonic function.*

Then ND-Λ finds solutions in time $O(\rho(|\Sigma_{\pi'_{\chi'}}|))$ in a nondeterminized version of that planning domain, where $|\Sigma_{\pi'_{\chi'}}|$ is the size of execution structure for the solution policy $\pi'_{\chi'}$ returned by ND-Λ.

Proof. Suppose Λ finds solutions in $O(\rho(|\pi_\chi|))$ time in strongly-connected classical planning domain. The following shows that ND-Λ finds solutions in time $O(\rho(|\Sigma_{\pi'_{\chi'}}|))$ in a nondeterminized version of that domain. Note that a nondeterminized version of a strong-connected classical planning domain is also strongly connected. Since the planning domains are strong-connected, for any state transition in $\Sigma_{\pi'_{\chi'}}$, there is a deterministic action $det(a)$ in P such that the intended effect of $det(a)$ induces that state transition. Therefore, each path in the execution structure $\Sigma_{\pi'_{\chi'}}$ is a solution plan for the classical planning problem P.

Suppose there are k paths in the execution structure $\Sigma_{\pi'_{\chi'}}$. In weak and strong planning, each of the paths in $\Sigma_{\pi'_{\chi'}}$ starts from an initial state s_0 and ends in a goal state s_g. In strong-cyclic planning, there may be some cyclic paths that start from an initial state and ends in a state that is visited before on that path. Note that as long as such cycles do not violate the "fairness assumption," the strong-cyclic policy $\pi'_{\chi'}$ that contains such cyclic paths is a solution.

Let $\langle s_0, s_1, s_2, \ldots, s_k \rangle$ be a cyclic path in $\Sigma_{\pi'_{\chi'}}$ where s_0 is an initial state and $s_k = s_i$ such that $i = 0, \ldots, k-1$ – i.e., s_k is a cyclic state. For each cyclic path p in $\Sigma_{\pi'_{\chi'}}$ we create

a classical planning problem $P'' = (s_0, \{s_k\}, \Sigma)$. For each acyclic path p in $\Sigma_{\pi'_{\chi'}}$, we create a classical planning problem $P'' = (s_0, \{s_g\}, \Sigma)$ where Σ is the classical planning domain Σ in which P was formulated, and s_g is the goal state in $\Sigma_{\pi'_{\chi'}}$ in which the path p ends. Now, suppose Λ returns p as a solution for P'' in time $O(\rho(|p|))$ then ND-Λ returns the solution policy $\pi'_{\chi'}$ in time $O(\rho(\Sigma_{\pi'_{\chi'}}))$ since, Lemma 3, the search space explored by ND-Λ is the same as that is explored by Λ for each such problem P'', and ND-Λ explores all the paths in $\Sigma_{\pi'_{\chi'}}$. ■

Corollary 5 *Under the conditions of Theorem 4, if the number of possible successors of each state is bounded by a constant, then ND-Λ finds solutions in time $O(\rho(|\pi'_{\chi'}|))$, where $|\pi'_{\chi'}|$ is the size of the solution policy.*

Proof. Let b be the number of possible successors of any state in a planning domain. Then, if the solution policy has size k, then the execution structure will have size $\leq bk$. ■

Theorem 6 *Suppose Λ finds solution plans in time $O(\rho(|\pi_\chi|))$ in a classical planning domain, given the search-control information χ. $|\pi_\chi|$ is the size of the solution plan and ρ is a monotonic function.*

Then, ND-Λ finds solutions in average time $O(\rho(n) + \frac{bdn}{t}))$, where $n = |\Sigma_{\pi'_{\chi'}}|$ is the size of the execution structure for the solution policy $\pi'_{\chi'}$ returned by ND-Λ given the search-control information χ', b the maximum number of state-action pairs that are added to any policy after ND-Λ generates a dead-end state-action pair, t is the maximum

number of actions applicable to a state, and in every state s, $0 \leq d \leq t$ is the maximum number of actions applicable to s that lead to a dead-end state.

Proof.

Suppose one of the invocations of ND-Λ generates a dead-end state-action pair (s, a). This means that one of the successor states s' generated by applying a in s does not have a π-descendant that is a goal state. There are two cases. The first case is straightforward: if s' does not have a π-descendant in the goal states or in the states of the $OPEN$ set of this invocation of ND-Λ, then the planning algorithm immediately returns FAILURE. In this case, the planning algorithm does not perform any additional work that is to be lost afterwards when it backtracks from this search trace at the point it detects that s' does not have any π-descendants that is a goal state.

Now, suppose that s' does have a π-descendant in the states of the $OPEN$ set of this invocation. Then ND-Λ will defer its decision on s' until all of those π-descendants of s' has been determined to be dead. The sub-policy π' that contains all of the state-action pairs, including (s, a), that are generated during this process process will be discarded during backtracking.

If this invocation of ND-Λ generates another dead-end state-action pair in this invocation after it backtracks and discards π', then ND-Λ will produce another π' and will discard at the end when it detects that π' is not a part of a solution. Suppose there are d possible state-action pairs that can be generated in this invocation and t of them are dead. Let b be the maximum size of all those dead policies. Then, ND-Λ will explore a search space of size $|\frac{bt}{d}|$, which is discarded at the end.

At each state in the solution policy $\pi'_{\chi'}$, ND-Λ could explore a redundant search space of size $|\frac{bt}{d}|$. The time complexity to generate the policy $\pi'_{\chi'}$ without exploring any dead-end state-action pairs is given by Theorem 4. Therefore, it follows that the time complexity for generating a solution is $O(\rho(n) + \frac{bdn}{t}))$, where b the maximum number of state-action pairs that are added to any policy after ND-Λ generates a dead-end state-action pair, t is the maximum number of actions applicable to a state, and in every state s, $0 \leq d \leq t$ is the maximum number of actions applicable to s that lead to a dead-end state. ■

Corollary 7 *Under the conditions of Theorem 6, if the number of possible successors of each state is bounded by a constant, then ND-Λ finds solutions in average time $O(\rho(|\pi'_{\chi'}|) + \frac{bd|\pi'_{\chi'}|}{t}))$, where $|\pi'_{\chi'}|$ is the size of the solution.*

Proof. Immediate from Theorem 6 and Corollary 5. ■

Corollary 8 *Suppose Λ finds solution plans in time $O(\rho(|\pi|))$ in a classical planning domain, where $|\pi|$ is the size of the solution plan and ρ is a monotonic function. Then, if the number of initial states in P' are bounded by a constant, ND-Λ returns weak solutions for nondeterminized versions of those planning problems in time $O(\rho(|\pi|))$.*

Proof. Immediate from the proof of Theorem 4. ■

A.2 Proofs for Chapter 4

Theorem 9 *The planning procedure FS^3 always terminates.*

Proof. The only possible situation in which FS^3 does not terminate is that the $OPEN$ set never becomes empty. However, this cannot happen because

- the state space of the planning problems are finite; and
- at the beginning of each invocation, the FS^3 removes the states that it already visited from the states of the $OPEN$ set. Therefore, FS^3 never visits a state more than once, and therefore, it does not caught in infinite search traces during planning.

Thus, it follows from the above that FS^3 always terminates. ■

Theorem 10 *Let $P = (\Sigma, S_0, G)$ be a planning problem in a nondeterministic planning domain Σ, and let π be a partial policy in Σ If π is a candidate solution for P, then an invocation of* NoGood(π, S_π^t, G, S_0) *returns* FALSE*, where S_π^t are the terminal states of π. Otherwise,* NoGood(π, S_π^t, G, S_0) *returns* TRUE.

Proof. The proof is in three parts for weak, strong, and strong-cyclic planning with NoGood function in FS^3. All of the subproofs are by contradiction.

Weak Planning.

Case 1. The NoGood function for weak planning traverses all of the execution paths in the execution structure Σ_π induced by π, performing successive the WeakPreimage operations. This is a backward search traversal that starts from the goal and non-goal terminal states of π towards the initial states of the planning problem P. At the end of this traversal, it computes the set S of states in π such that there exists at least one path from a state s in S to a goal or a non-goal terminal state.

Suppose π is a candidate weak solution for P. Assume that the invocation NoGood(π, S^t_π, G, S_0) function returns TRUE. The only case in which NoGood returns TRUE is when it traverses all paths in Σ_π and detects that there exists at least one initial state s_0 in S_0 such that s_0 is not in S. This means that there is no execution path in the execution structure Σ_π that starts from s_0 and ends in a goal state or in a non-goal terminal state in S^t_π. However, by the definition of candidate solutions, this is a contradiction. Therefore, if π is a candidate solution then NoGood never returns TRUE.

Case 2. Now suppose π is not a candidate weak solution and NoGood(π, S^t_π, G, S_0) function returns FALSE. NoGood returns FALSE only if for each initial state s_0 in S_0, there exists at least one path in the execution structure Σ_π that starts in s_0 and ends in a goal or non-goal terminal state. However, since π is not a candidate solution, there must at least one initial state for which this does not hold. Therefore, NoGood does not return FALSE for π, which is a contradiction.

Strong Planning.

Case 1. The NoGood function for strong planning is similar to that for weak planning, but it traverses all of the execution paths in the execution structure Σ_π induced by π, performing successive the StrongPreimage operations. This is a backward search traversal that starts from the goal and non-goal terminal states of π towards the initial states of the planning problem P. At the end of this traversal, it computes the set S of states in π such that all of the paths from a state s in S reaches to a goal or a non-goal terminal state in the execution structure Σ_π. During the backward search, NoGood removes the state-action pairs that it visits from the partial policy π. At the end of the traversal, if there are state-

action pairs left in π then there is a cyclic path in the execution structure Σ_π since this means that the state s left in pi has a successor state s' such that s' is a $\pi-$ ancestor of s. In this case, NoGood returns TRUE.

Suppose π is a candidate strong solution for P. Assume that the invocation NoGood(π, S^t_π, G, S_0) function returns TRUE. There are only two cases in which NoGood returns TRUE. First is when it traverses all paths in Σ_π and detects that there exists at least one initial state s_0 in S_0 such that s_0 is not in S. This means that there is no execution path in the execution structure Σ_π that starts from s_0 and ends in a goal state or in a non-goal terminal state in S^t_π. However, by the definition of candidate solutions, this is a contradiction. Secondly, the backward traversal of π does not remove all of the state-action pairs from π, as described above. In this case, there is a cyclic path in the execution structure Σ_π, which means that π is not a candidate strong solution.

Therefore, if π is a candidate strong solution then NoGood never returns TRUE.

Case 2. Now suppose π is not a candidate strong solution and NoGood(π, S^t_π, G, S_0) function returns FALSE. NoGood returns FALSE only if (1) for each initial state s_0 in S_0, there exists at least one path in the execution structure Σ_π that starts in s_0 and ends in a goal or non-goal terminal state, and (2) the backward search removes all of the state-action pairs from π. However, since π is not a candidate solution, π must be violating one or both of these properties. Therefore, NoGood does return TRUE for π, which is a contradiction.

Strong-Cyclic Planning. The proof for this case is the same as the one for the strong-planning case, except that the NoGood function returns TRUE for cyclic paths that do no

have any possibility to reach to the goals. ■

Theorem 11 *Suppose one of the search traces of FS^3 returns a policy π given the input planning problem $P = (\Sigma, S_0, G)$ in a nondeterministic planning domain Σ. Then π is a solution for the planning problem P.*

Proof. The proof is by contradiction. Suppose one of the search traces of FS^3 returns a policy π for P. Assume that π is not a solution for P. This means that there exists at least one invocation of FS^3 where the partial policy, say π', in that invocation is not a candidate solution. However, since FS^3 did not return FAILURE, its NoGood function must have returned FALSE in all invocations of FS^3 on this search trace. However, by Theorem 10, this is not possible: NoGood returns TRUE for an input partial policy that is not a candidate solution. Therefore, π must be a solution for the input planning problem P.

A remark regarding the above proof is in order. Note that, in any invocation of FS^3, the states in the $OPEN$ set (i.e., $S = \mathsf{StatesOf}(OPEN)$), constitute exactly the set S^t_π of non-goal terminal states of the partial policy in that invocation. Hence, the application of the Theorem 10 above. ■

Theorem 12 *Suppose $P = (\Sigma, S, G)$ is a χ-solvable nondeterministic planning problem given the search-control information χ. Then, one of the search traces of FS^3 returns a solution policy for P using χ.*

Proof. We define the *minimum solution size* for the planning problem P as $|S_\pi|$, where S_π is the set of states in a solution policy π, such that there is no other solution policy π'

for P such that $|S_{\pi'}| < |S_{\pi}|$. The proof is by induction on n, the minimum solution size for P:

Base Step ($n = 0$). In this case, the solution policy $\pi = \emptyset$. This means that $S_0 \subseteq G$. Indeed, in its initial invocation, FS^3 removes all of the states from the only situation in $OPEN$ and leaves $OPEN$ as the empty set. As a result, it returns the empty policy as a solution.

Induction Step. Let $n > 1$ and suppose the theorem is true for every $k < n$. Note that this means that this invocation of FS^3 is attempting to solve the planning problem P such that S is the set of all states in $OPEN$. Then, in this invocation of FS^3, the following must be true:

- there must be a situation (S, χ) in $OPEN$ for which there must be a non-empty set F of tuples of the form $(S \cap S_a, a, \chi')$ generated by using the search-control function acceptable the search-control formula χ, such that F specifies one and only one action for each state in S.

- Let $OPEN'$ be the same as $OPEN$, plus it includes the successor situations of F as well. Let S' be the set of all states in $OPEN'$. Then, the minimum solution size for the planning problem (S', G, Σ) must be $n - |F|$ since, otherwise, P could not have the minimum solution size n.

Then, one of the nondeterministic search traces in FS^3 generates the set F and recursively invokes itself for the planning problem (S', G, Σ). By the induction hypothesis, this recursive invocation of the procedure generates a solution policy π' for (S', G, Σ). During this process, FS^3 combines the policy π' with the current partial policy computed

so far in the forward search, and it returns the solution policy π. ■

A.3 Proofs for Chapter 5

Theorem 13 *Suppose Z returns a solution policy π for P. Then, Z^F returns a solution policy π' for P such that $V_\pi(s) = V_{\pi'}(s)$ for every $s \in S_0$, if $\mathsf{acceptable}_F$ is admissible for P.*

Proof. Without loss of generality, suppose there is only one initial state (i.e., S_0 is a singleton set). Let π be a solution policy returned by the original MDP algorithm Z. Then, by the definition of a solution for an MDP planning problem, we have

$$V_\pi(s_0) = V(s_0) = C(s_0, a) + \alpha \sum_{s' \in \gamma(s_0,a)} Pr(s_0, a, s')V(s'),$$

where a is the action specified by the policy π. By the admissibility of $\mathsf{acceptable}_F$, the above is also true in the reduced MDP Σ^F.

Now, suppose that the original planning algorithm Z returns a solution policy π for P. Suppose the enhanced MDP planning algorithm Z^F returns a policy π' for P, using an admissible search-control function $\mathsf{acceptable}_F$. Assume that $V_\pi(s_0) \neq V_{\pi'}(s_0)$. There are two cases:

- $V_\pi(s_0) > V_{\pi'}(s_0)$. In this case, the value of the initial state in the reduced MDP Σ^F is less than that of the same state in the original MDP Σ. This means that the solution π' in the reduced case is not among the solutions for the planning problem P in the original MDP. This could only happen if in the reduced MDP, there are

new actions introduced by $\mathsf{acceptable}_F$ that do not appear in the original MDP, and those actions are a part of the solutions in the reduced MDP. However, this is not possible since the set of $\mathsf{acceptable}$ actions in a state s is always a subset of the set of applicable actions in s.

- $V_\pi(s_0) < V_{\pi'}(s_0)$. This means that there is a state s in π' such that

$$V(s) < V_{\pi'}(s) = C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V_{\pi'}(s'),$$

where a is the action specified in π' for s and $V(s) = \min_{a \in \mathsf{app}(s)} C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V(s')$ since the algorithm Z considers all of the actions in $\mathsf{app}(s)$.

Let $X(s)$ be the set of actions for which $\mathsf{acceptable}_F$ holds in the state s. Note that $X(s)$ is a subset of $\mathsf{app}(s)$ (i.e., $X(s) \subseteq \mathsf{app}(s)$) by the definition of $\mathsf{acceptable}_F$. Then, we have the following,

$$\begin{aligned} V(s) = & \min_{a \in \mathsf{app}(s)} C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V(s') \\ < & \quad C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V_{\pi'}(s') \\ = & \min_{a \in X(s)} C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V(s') \end{aligned}$$

From this, it follows that for each action a in $X(s)$, we have

$$V(s) < C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s') V(s'),$$

and thus,

$$V(s) \neq C(s,a) + \alpha \sum_{s' \in \gamma(s,a)} Pr(s,a,s')V(s').$$

However, this contradicts with the admissibility of $\mathsf{acceptable}_F$.

Therefore, if $\mathsf{acceptable}_F$ is admissible, $V_{\pi'}(s_0)$ must be equal to $V_{\pi}(s_0)$, ■

BIBLIOGRAPHY

[ACG+01] L. C. Aiello, A. Cesta, E. Giunchiglia, M. Pistore, and P. Traverso. Planning and verification techniques for the high level programming and monitoring of autonomous robotic devices. In *Proceedings of the European Space Agency Workshop on On Board Autonoy*, Noordwijk, Netherlands, October 2001. ESA.

[ACGT01] L. C. Aiello, A. Cesta, E. Giunchiglia, and P. Traverso. Merging Planning and Verification Techniques for "Safe Planning" in Space Robotics. In *6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey (ISAIRAS01)*, Montreal, Canada, June 2001.

[AD94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183—235, 1994.

[Bac01] Fahiem Bacchus. The AIPS '00 planning competition. *AI Magazine*, 22(1):47–56, 2001.

[BBS95] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[BCP+01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information*, Seattle, USA, August 2001.

[BCRT01] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 473–478, Seattle, USA, August 2001. Morgan Kaufmann.

[BCRT06] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Strong Planning under Partial Observability. *Artificial Intelligence*, 170:337–384, 2006.

[BDG00] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[BDH99] Craig Boutilier, Thomas L. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR*, 11:1–94, 1999.

[Bel57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Ber05] D.P. Bertsekas. *Dynamic Programming and OptimalControl*, volume 2. Athena Scientific, 2005.

[BF97] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[BG96] Craig Boutilier and Moises Goldszmidt. The frame problem and bayesian network action representation. In *Canadian Conference on AI*, pages 69–83, 1996.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning (ECP)*, pages 360–372, Durham, UK, 1999. Springer-Verlag.

[BG00] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In Steve Chien, S. Kambhampati, and C.A. Knoblock, editors, *Proceedings of the International Conference on AI Planning Systems (AIPS)*, pages 52–61. AAAI Press, April 2000.

[BG01a] B. Bonet and H. Geffner. GPT: a tool for planning with uncertainty and partial information. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 82–87, 2001.

[BG01b] B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

[BG03] B. Bonet and H. Geffner. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In E. Giunchiglia, N. Muscettola, and D. Nau, editors, *ICAPS-03*, pages 12–21, 2003.

[BK00] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[BK04] Daniel Bryce and Subbarao Kambhampati. Heuristic Guidance Measures for Conformant Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2004.

[BL99] A.L. Blum and J.C. Langford. Probabilistic planning in the Graphplan framework. In *Proceedings of the European Conference on Planning (ECP)*, pages 319–332, 1999.

[BRP01] Craig Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BT96] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[CCP+99] A. Chiappini, A. Cimatti, C. Porzia, G. Rotondo, R. Sebastiani, P. Traverso, and A. Villafiorita. Formal Specification and Development of a Safety-Critical Train Management. In *Proceedings of the 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP99)*, volume 1698 of *Lecture Notes in Computer Science (LNCS)*, pages 410–419, Toulouse, France, September 1999. Springer-Verlag.

[CFHM05] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. An adaptive samping algorithm for solving markov decision processes. *Oper. Res.*, 53(1):126–139, 2005.

[CGGT97] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for AR. In *Proceedings of the European Conference on Planning (ECP)*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 130–142, Toulouse, France, September 1997. Springer-Verlag.

[CGM+97] A. Cimatti, F. Giunchiglia, G. Mongardi, B. Pietra, D. Romano, F. Torielli, and P. Traverso. Formal Validation & Verification of Software for Railway Control and Protection Systems: Experimental Applications in ANSALDO. In *Proc. World Congress on Railway Research (WCRR'97)*, volume C, pages 467–473, Firenze, Italy, November 1997.

[CGM+98] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal Verification of a Railway Interlocking System using Model Checking. *Journal on Formal Aspects in Computing*, 10(4):361–380, 1998. Springer.

[CGT03] C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1–2):85–117, 2003.

[CKL94] A. Cassandra, Leslie Pack Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 1994.

[CLRS01] T.H. Cormen, C.E. Leirson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[CPS+99] A. Cimatti, P.L. Pieraccini, R. Sebastiani, P. Traverso, and A. Villafiorita. Formal Specification and validation of a Vital Communication Protocol. In *Proceedings of the World Congress on Formal methods in the Development of Computing Systems (FM'99)*, volume 1709 of *Lecture Notes in*

Computer Science (LNCS), pages 1584–1604, Toulouse, France, September 1999. Springer-Verlag.

[CRT98a] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI/IAAI Proceedings*, pages 875–881, 1998.

[CRT98b] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, pages 36–43. AAAI Press, June 1998.

[CT91] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[DB97] Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1-2):219–283, 1997.

[dFVR03] D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Oper. Res.*, 51(6):850–865, 2003.

[Die00] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.

[DKKN93] Thomas L. Dean, Leslie Pack Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 574–579, 1993.

[DKKN95] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, July 1995.

[DPT02] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *AAAI/IAAI Proceedings*, pages 447–454, Edmonton, Canada, August 2002. AAAI Press/The MIT Press.

[DTV99] M. Daniele, P. Traverso, and M. Vardi. Strong cyclic planning revisited. In *Proceedings of the European Conference on Planning (ECP)*, pages 35–48, September 1999.

[EDMM03] E. Even-Dar, S. Mannor, and Y. Mansour. Action elimination and stopping conditions for reinforcement learning. In *Proceedings of the 20th International Conference on Machine Learning (ICML-2003), Washington DC*, 2003.

[ENS95] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

[FG00] P. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *AAAI/IAAI Proceedings*, pages 748–753. AAAI Press, 2000.

[FH02] Z. Feng and E. Hansen. Symbolic Heuristic Search for Factored Markov Decision Processes. In *AAAI-2002*, 2002.

[FL02] Maria Fox and Derek Long. International planning competition, 2002. http://www.dur.ac.uk/d.p.long/competition.html.

[GB94] R. P. Goldman and M. S. Boddy. Conditional linear planning. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, 1994.

[Giu00] E. Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, 2000.

[GK03] Natalia H. Gardiol and Leslie Pack Kaelbling. Envelope-based planning in realtional MDPs. In *Advances in Neural Information Processing Systems*, 2003.

[GKP01a] C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored mdps. In *NIPS*, 2001.

[GKP01b] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projections for factored MDPs. In *IJCAI*, pages 673–682, 2001.

[GMP00] R. P. Goldman, D. J. Musliner, and M. J. Pelican. Using model checking to plan hard real-time controllers. In *Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, Breckenridge, Colorado, April 2000.

[GN92] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

[GN93] M. Genesereth and I. Nourbakhsh. Time-saving tips for problem solving with incomplete information. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1993.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[GPM99] R.P. Goldman, M. Pelican, and D.J. Musliner. Hard real-time mode logic synthesis for hybrid control: A CIRCA-based approach, mar 1999. Working notes of the 1999 AAAI Spring Symposium on Hybrid Control.

[GT99] F. Giunchiglia and P. Traverso. Planning as model checking. In *Proceedings of the European Conference on Planning (ECP)*, pages 1–20, September 1999.

[HBG05] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2005.

[HG00] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, pages 140–149, 2000.

[HK04] M. Hauskrecht and B. Kveton. Linear Program Approximations for Factored Continuous-State MDPs. In *NIPS-2004*, 2004.

[HM93] Steve Hanks and Drew McDermott. Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. Technical Report TR-93-06-10, University of Washington, Department of Computer Science and Engineering, 1993.

[HN01] J. Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Potomac, MD, 1978.

[HS94] P. Haddawy and M. Suwandi. Decision-theoretic refinement planning using inheritance abstraction. In *AIPS-1994 Proceedings*, 1994.

[HSAHB99] J. Hoey, R. St-Aubin, A. Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.

[HSMN96] Kiran Hebbar, Stephen J. J. Smith, I. Minis, and Dana S. Nau. Plan-based evaluation of designs for microwave modules. In *Proc. ASME Design Technical Conference*, August 1996.

[HZ98] E. A. Hansen and S. Zilberstein. Heuristic search in cyclic and-or graphs. In *AAAI/IAAI Proceedings*, 1998.

[HZ01] E. Hansen and S. Zilberstein. LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops. *Artificial Intelligence*, 129:35–62, 2001.

[INMA06] Okhtay Ilghami, Dana S. Nau, and Hector Muñoz-Avila. Learning to do HTN planning. In *Proceedings of the Sixteenth International Conference on AI Planning and Scheduling*, Cumbria, UK, June 2006. AAAI Press.

[INMAA05] Okhtay Ilghami, Dana S. Nau, Hector Muñoz-Avila, and David W. Aha. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence*, 21(4):388–413, november 2005.

[JV00] R. Jensen and Manuela M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *JAIR*, 13:189–226, 2000.

[JVB01] R. Jensen, Manuela M. Veloso, and M. H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the European Conference on Planning (ECP)*, 2001.

[JVB03] R. Jensen, Manuela M. Veloso, and R.E. Bryant. Guided symbolic universal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, June 2003. AAAI Press.

[Kar01] L. Karlson. Conditional progressive planning under uncertainty. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[KBSD97] Froduald Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[KD01] Jonas Kvarnström and Patrick Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Articial Intelligence*, 30:119–169, 2001.

[KHW94] N. Kushmerick, S. Hanks, and D. S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1994.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[KN04] Ugur Kuter and Dana Nau. Forward-chaining planning in nondeterministic domains. In *AAAI-2004*, 2004.

[Kno94] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[Koe99] Jana Koehler. Handling of conditional effects and negative goals in IPP. Technical note 128, Freiburg Univ., 1999.

[Kor90] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211, 1990.

[KP99] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *IJCAI*, pages 1332–1339, 1999.

[KP00] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *UAI*, pages 326–334, 2000.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 359–363, 1992.

[KS95] Sven Koenig and Reid G. Simmons. Real-time search in non-deterministic domains. In *IJCAI-1995*, 1995.

[KS98a] H. Kautz and B. Selman. Blackbox: A sat-technology planning system, 1998.

[KS98b] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, 1998.

[LC06] P. Langley and D. Choi. Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, 7:493–518, 2006.

[Lit97] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *AAAI/IAAI Proceedings*, pages 748–761, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[LK02] T. Lane and L.P. Kaelbling. Nearly deterministic abstractions of Markov decision processes. In *AAAI-2002*, 2002.

[LTS99] J. Lygeros, C. Tomlin, and Shankar Sastry. Controllers for Reachability Specifications for Hybrid Systems. *Automatica*, 35(3), March 1999.

[LY04] M. Littman and H.L.S. Younes. The probablistic planning track of the 2004 international planning competition, 2004. http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/.

[Mac66] J. MacQueen. A modified dynamic programming method for markovian decision problems. *J. Math. Anal. Appl.*, 14:38–43, 1966.

[NAI+03] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *JAIR*, 20:379–404, December 2003.

[NAI+05] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Muoz-Avila, J. W. Murdock, D. Wu, and F. Yaman. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, March–April 2005.

[Nil80] Nils Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[NK01] N. Nguyen and Subbarao Kambhampati. Reviving partial order planning. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[NKN02] N. Nguyen, Subbarao Kambhampati, and R. Nigenda. Planning graph as the basis for deriving heuristics for plan synthsis by state space and CSP search. *Artificial Intelligence*, 2002.

[OP99] N. Onder and M. E. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 577–584, 1999.

[Par98] R. Parr. Hierarchical control and learning for markov decision processes, 1998. PhD thesis, UC Berkeley.

[PB00] Pascal Poupart and Craig Boutilier. Value-directed belief state approximation for pomdps. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 497–506, 2000.

[PB01] Pascal Poupart and Craig Boutilier. Vector-space analysis of belief-state approximation for pomdps. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 445–452, 2001.

[PB02] R. Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, 2002.

[PBT01] M. Pistore, R. Bettin, and P. Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Proceedings of the European Conference on Planning (ECP)*, 2001.

[PC96] L. Pryor and G. Collins. Planning for contingency: a decision based approach. *Journal of Artificial Intelligence Research*, 4:81–120, 1996.

[PPS+02] R. Patrascu, P. Poupart, D. Schuurmans, C. Boutilier, and C. Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *AAAI/IAAI Proceedings*, 2002.

[PS92] M. Peot and D. Smith. Conditional nonlinear planning. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, pages 189–197, 1992.

[PSP94] S. Panda, F. Somenzi, and B. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 628–631, 1994.

[PT01] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 479–484, Seattle, USA, August 2001. Morgan Kaufmann.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley & Sons, Inc., New York, 1994.

[PW92] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for adl. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, 1992.

[Rin99a] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[Rin99b] J. Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1192–1197, Stockholm, Sweden, 1999. Morgan Kaufmann.

[Rin02] J. Rintanen. Backward plan construction for planning as search in belief space. In *AIPS-2002*, 2002.

[Rin05] J. Rintanen. Conditional planning in the discrete belief space. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[RN03] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach (Second Edition)*. Prentice-Hall, Upper Saddle River, NJ, 2003.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

[SB98] R. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[SNT98] Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[Son78] E. J. Sondik. The Optimal Control of Partially Observable Markov Decision Processes. *Operation Research*, 26(2):282–304, 1978.

[SP01] D. Schuurmans and R. Patrascu. Direct value-approximation for factored MDPs. In *Proc. NIPS-14*, 2001.

[SPS00] O. Shakernia, G.J. Pappas, and S.S. Sastry. Decidable Controller Synthesis for Classes of Linear Systems. In *Hybrid Systems: Computation and Control (LNCS 1790)*, 2000.

[ST01] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, January 2001.

[SW98] David E. Smith and Daniel S. Weld. Conformant Graphplan. In *AAAI/IAAI Proceedings*, pages 889–896, July 26-30 1998.

[TLS00] C. Tomlin, J. Lygeros, and Shankar Sastry. A Game Theoretic Approach to Controller Design for Hybrid Systems. *IEEE Proceedings*, 88(7), July 2000.

[TMBO03] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi. Computational Techniques for the Verification and Control of Hybrid Systems. *IEEE Proceedings*, 91(7), July 2003.

[TVR96] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large-scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[TZ97] M. Trick and S. Zin. Spline approximations to value functions: A linear programming approach. *Macroeconomic Dynamics*, 1:255–277, 1997.

[WAS98] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *AAAI/IAAI Proceedings*, pages 897–904, Menlo Park, July 26–30 1998. AAAI Press.

[Wat89] C.J.C.H. Watkins. *Learning from delayed rewards*. Ph.d. dissertation, Kings College, 1989.

[Wil88] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[Wil90] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.

[YMS03] Håkan L. S. Younes, David J. Musliner, and Reid G. Simmons. A framework for planning in continuous-time stochastic domains. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, pages 195–204, Trento, Italy, June 2003. AAAI Press.

[YS02] H. Younes and Reid Simmons. *On the role of ground actions in refinement planning*, pages 54–61. AAAI Press, 2002.